TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT MATHEMATIK UND NATURWISSENSCHAFTEN

INSTITUT FÜR ALGEBRA

The Foundation of Pattern Structures and their Applications

DISSERTATION

ZUR ERLANGUNG DES AKADEMISCHEN GRADES

DOCTOR RERUM NATURALIUM
(DR. RER. NAT.)

EINGEREICHT AM: 09. MÄRZ 2021
VERTEIDIGT AM: 24. SEPTEMBER 2021

Autor: Dipl.-Math. Lars LUMPE

Betreuer und Erstgutachter: Prof. Dr. Stefan E. SCHMIDT

Zweitgutachter: Prof. Dr. Sergei KUZNETSOV

Danksagung

Ich danke Stefan Schmidt für die Möglichkeit, diese Arbeit zu schreiben, aber noch viel dankbarer bin ich für seine stetige Unterstützung bei der Erstellung dieser Arbeit zu jeder Tages- und Nachtzeit. Es war nicht nur aus mathematischer Sicht eine sehr lehrreiche Zeit. Auch allen Korrekturlesern möchte ich für Ihre Zeit und Geduld danken. Weiterhin möchte ich meinen Eltern Detlef und Heike Lumpe dafür meinen Dank aussprechen, dass sie mir schon mein ganzes Leben lang auf jede erdenkliche Art und Weise unter die Arme greifen. Vielen Dank, dass ihr immer für mich da seid. Nicht fehlen darf natürlich Linda, die mir den Freiraum ermöglicht hat diese Arbeit zu schreiben, mir mit Rat und Tat zur Seite steht und auch das Zusammenleben mit mir irgendwie meistert. Lias Albert möchte ich danken, dass er mich immer wieder an das wirklich Wichtige im Leben erinnert. Ich freue mich darauf, dir eines Tages diese Arbeit zu zeigen. Ich bin gespannt, was du dazu sagst.

–There are only patterns, patterns on top of patterns, patterns that affect other patterns. Patterns hidden by patterns. Patterns within patterns. If you watch close, history does nothing but repeat itself. What we call chaos is just patterns we haven't recognized. What we call random is just patterns we can't decipher.

- Chuck Palahniuk, *Survivor (1999)*

Contents

Preface

This thesis is based on the publications [LS15a], [LS15b], [LS16], [LS17], [LS20a] and [LS20b]. Pattern structures were introduced in [GK01] and the literature suggested, that projections of pattern structures lead again to pattern structures (for example see [GK01, Kuz09]). In Chapter 3 ([LS15a]) we give a counter example, where it turns out that projections do not always give rise to new pattern structures. We also provide a solution to the problem: residual projections. They lead again to pattern structures. As a response to our paper [LS15a], another solution, called o-projections, was introduced in [BKN15b]. In [LS15b], which is represented in Chapter 4, we then showed that residual projections and o-projections are linked. We introduced pattern morphisms, which are a useful tool to describe connections between pattern structures. The idea has been picked up and so interesting publications like [Bel19a], [Bel19b], [BKK18], [BBK19], [BKK20] and [CCOGP+16] originated and reference to our paper. To complete the theory on this subject, we investigate the impact of morphisms between pattern structures on concept lattices [LS16] and on their representations [LS17]. In this thesis, these topics are discussed in Chapter 5 and 6.
The application part essentially consists of two ideas. On the one hand, we show that decision trees and random forests, introduced in [BFSO84] respectively in [Bre01], which are very useful tools for many real world data mining problems (for example see [Mah05], [DUA06], [SLT+03], [BD16], [RGGR+12]), can be described via pattern structures. This is the subject of Chapter 7, where we also demonstrate that this is not just a nice view on things, but also gives rise to new pattern structures which can be used as a data mining tool. We connected the work on decision trees and random forests to pattern structures and particularly to interval pattern structures, which were introduced in [KKND11]. On the other hand, in Chapter 8, we build a pattern structure and used clustering algorithms to search for important patterns in it. For both ideas in Chapter 7 and 8 we give a first real world application by building a model for a classification problem of red wines. These are first examples that our ideas are useful. Further investigations are needed to prove the importance of the presented methods and build algorithms, which can be used on other datasets and use cases.

Part I

Theory

1 Preliminaries

1.1 Order-theoretical Foundations

This section contains definitions and theorems, laying the foundation for the following chapters and allowing for a better understanding of the subject of this thesis. The main parts of this chapter are taken from [GW13].

Definition 1.1 (binary relation)**.** A binary **relation** R between two sets M and N is a set of pairs (m,n) with $m \in M$ and $n \in N$, i.e., a subset of the set $M \times N$ of all such pairs. R^{-1} or R^d denotes the **inverse** or **dual relation** to R, that is the relation between N and M with $nR^d m :\Leftrightarrow mRn$. If $N = M$, we speak of a **binary relation** on the set M.

Definition 1.2 (relation, order, poset)**.** A binary relation R on a set M is called:

1.) **reflexive**, if xRx for all $x \in M$

2.) **transitive**, if for all $x,y,z \in M$ with xRy and yRz it follows that xRz

3.) **symmetric**, if xRy it follows that yRx

4.) **antisymmetric**, if xRy and yRx it follows that $x = y$.

A binary relation R on a set M is called **equivalence relation** on M, if R is reflexive, transitive and symmetric. We call a binary relation R on a set M **order relation** or short an **order** on M, if R is reflexive, transitive and antisymmetric. A **partially ordered set (poset)** $\mathbb{M} := (M, \leqslant)$ is a pair $(M, \leqslant)$, with a set M and an order relation $\leqslant$ on M.

Remarks:

(1) Let $\mathbb{P} = (P, \leqslant)$ be a poset and $T \subseteq P$. Then the **restriction** of $\mathbb{P}$ onto T is given by

$$\mathbb{P}|T := (T, \leqslant \cap (T \times T)),$$

which clearly is a poset too.

(2) For every set M the **power set** of M, which is the set of all subsets of M with the set inclusion as order, is an ordered set. We denote the power set of M by 2^M.

Definition 1.3 (chain, antichain)**.** Two elements of an ordered set $(M,\leqslant)$ are called **comparable** if $x \leqslant y$ or $y \leqslant x$; otherwise they are **incomparable**. A subset of $(M,\leqslant)$ in which any two elements are compareable is called a **chain**; a subset in which any two elements are incompareable is called an **antichain** .

Definition 1.4 (principal ideal, principal filter)**.** Let $\mathbb{P} := (P,\leqslant)$ be a poset. We call a set $X \subseteq P$ **principal ideal** if for every $x \in X$ and $z \leqslant x$ it follows that $z \in X$. On the other hand, we call a set $Y \subseteq P$ **principal filter** if, for every $y \in Y$ and $y \leqslant z$, it follows that $z \in Y$.

Definition 1.1 (isotone and antitone maps)**.** Let $\mathbb{P} := (P,\leqslant)$, $\mathbb{L} := (L,\leqslant)$ be ordered sets and $f : P \to L$ a map; then we call f **isotone** if for all $x,y \in P$ with $x \leqslant y$ it follows that

$$f(x) \leqslant f(y).$$

We call the map **antitone** if for all $x,y \in P$ with $x \leqslant y$ it follows that

$$f(y) \leqslant f(x).$$

Definition 1.5 (infimum, supremum)**.** Let $(M,\leqslant)$ be an ordered set and A a subset of M. A **lower bound** of A in $(M,\leqslant)$ is an element s of M with $s \leqslant a$ for all $a \in A$. An **upper bound** of A in $(M,\leqslant)$ is an element s' of M with $a \leqslant s'$ for all $a \in A$. If there is a largest element in the set of all lower bounds of A, it is called the **infimum** of A and denoted by $\inf A$ respectively $\inf_{a\in A} a$. A least upper bound is called **supremum** and denoted by $\sup A$ or $\sup_{a\in A} a$. If $A = \{x,y\}$, we also write $x \wedge y$ for $\inf A$ and $x \vee y$ for $\sup A$. In the following we want to introduce our notation for an index set I and $\alpha \in M^I$, that is $\alpha : I \to M$:

$$\inf_{i\in I} \alpha i := \inf \alpha := \inf \alpha I.$$

For sup the notation is analogous.

Definition 1.6 (lattice, complete lattice)**.** An ordered set $\mathbb{V} := (V,\leqslant)$ is a **lattice**, if for any two elements x and y in V the supremum $x \vee y$ and the infimum $x \wedge y$ always exist. $\mathbb{V}$ is called a **complete lattice**, if $\sup X$ and $\inf X$ exist for any subset X of V.

Definition 1.7 (duality principle)**.** The inverse relation $\leqslant^d$ or $\geqslant$ of an order relation $\leqslant$ is also an order relation. It is called the **dual order** of $\leqslant$. We obtain the dual statement A^d of an order theoretic statement A (which, apart from purely logical components, only contains the symbol $\leqslant$) if we replace in A the symbol $\leqslant$ by $\geqslant$. A holds for an ordered set, if and only if A^d holds for the dual ordered set. This Duality Priciple is used to simplify definitions and proofs. If a theorem asserts two statements that are dual to each other, one follows "dually" i.e., with the same proof for the dual order.

Remark: For a poset $\mathbb{M} := (M, \leqslant)$, let $\mathbb{M}^d := (M, \leqslant^d)$ with

$$M^d := \{(y,x) \mid (x,y) \in \leqslant\}$$

denote the dual of $\mathbb{M}$.

Definition 1.8 (closure operator)**.** A **closure operator** on a poset $\mathbb{P} = (P, \leqslant)$ is a map $h : P \to P$, such that $\forall x, y \in P$:

$$x \leqslant hy \Leftrightarrow hx \leqslant hy \tag{1.1}$$

A subset C of P is called a **closure system** in $\mathbb{P}$ if for every $x \in P$ the restriction of $\mathbb{P}$ onto $\{t \in C \mid x \leqslant t\}$ has a least element.

Conclusion 1.1. Let $\mathbb{P} = (P, \leqslant)$ be a poset. If and only if $\forall x, y \in P$ the following conditions are fulfilled, the map $h : P \to P$ is a closure operator:

1.) $x \leqslant hx$ (extensive)

2.) $x \leqslant y \Rightarrow hx \leqslant hy$ (isotone)

3.) $h(hx) = hx$ (idempotent)

Proof. To show: Equivalence 1.1 $\Leftrightarrow$ (extensive), (isotone), (idempotent)

"$\Rightarrow$":

1. It applies: $hx \leqslant hx \overset{1.1}{\Rightarrow} x \leqslant hx$
2. Let $x \leqslant y$. 1. implies $y \leqslant hy$ and therefore $x \leqslant hy \overset{1.1}{\Rightarrow} hx \leqslant hy$
3. Inserting $y := hx$ into the Equivalence 1.1 leads to: $x \leqslant h(hx) \Leftrightarrow hx \leqslant h(hx)$. Putting $x := hy$ into Equivalence 1.1 result in $hy \leqslant hy \Leftrightarrow h(hy) \leqslant hy$. Therefore, variation of variables leads to $h(hx) = hx$.

"⇐":

"⇒": $x \leqslant hy \overset{2.}{\Rightarrow} hx \leqslant h(hy) \overset{3.}{\Rightarrow} hx \leqslant hy$.
"⇐": Follows immediately from 1.

□

Definition 1.9 (kernel operator)**.** A **kernel operator** on a poset $\mathbb{P} := (P, \leqslant)$ is a map $h : P \to P$ such that for all $x, y \in P$:

$$kx \leqslant y \Leftrightarrow kx \leqslant ky \tag{1.2}$$

A subset K of P is called a **kernel system** in $\mathbb{P}$ if for every $x \in P$ the restriction of $\mathbb{P}$ onto $\{t \in K \mid t \leqslant x\}$ has a greatest element.

Remark: A closure operator on $\mathbb{P} := (P, \leqslant)$ is defined as a kernel operator on $\mathbb{P}^d$, and a closure system in $\mathbb{P}$ is defined as a kernel system in $\mathbb{P}^d$.

Conclusion 1.2. Let $\mathbb{P} = (P, \leqslant)$ be a partially ordered set. If and only if $\forall x, y \in P$ the following conditions are fulfilled, the map $h : P \to P$ is a kernel operator:

1.) $kx \leqslant x$ (intensive)
2.) $x \leqslant y \Rightarrow kx \leqslant ky$ (isotone)
3.) $k(kx) = kx$ (idempotent)

Proof. Dual to Conclusion 1.1.

□

Lemma 1.1. If $KO\mathbb{P}$ denotes the set of all kernel operators on $\mathbb{P} := (P, \leqslant)$ and $KS\mathbb{P}$ denotes the set of all kernel systems in $\mathbb{P}$, then the map

$$KO\mathbb{P} \to KS\mathbb{P}, \psi \mapsto \psi P$$

is a bijection. Also, if $CO\mathbb{P}$ denotes the set of all closure operators on $\mathbb{P}$ and $CS\mathbb{P}$ denotes the set of all closure systems in $\mathbb{P}$, then

$$CO\mathbb{P} \to CS\mathbb{P}, \psi \mapsto \psi P$$

is a bijection.

Proof. For every kernel system $K \in KS\mathbb{P}$, the mapping

$$p \mapsto \sup\{k \in K \mid k \leqslant p\}$$

is a kernel operator. On the other hand, the image $\{\psi x \mid x \in P\}$ of a given kernel operator $\psi \in KO\mathbb{P}$ is a kernel system. Furthermore, the two transformations are inverse to each other and so describe a one-to-one correspondence between kernel systems and kernel operators. The other statement follows dually.

□

In the following we are going to introduce the concept of an adjunction [Ern04, see p.5], which is one of the most important definitions for our work.

Definition 1.10 (adjunction)**.** Let $\mathbb{P} = (P, \leqslant_{\mathbb{P}})$ and $\mathbb{L} = (L, \leqslant_{\mathbb{L}})$ be posets; furthermore, let

$$f : P \to L$$
$$\text{and } g : L \to P$$

be maps. We call (f, g) an **adjunction** w.r.t. $(\mathbb{P}, \mathbb{L})$ or $(\mathbb{P}, \mathbb{L}, f, g)$ a **poset adjunction** if for all $x \in P$ and $y \in L$ the following holds:

$$fx \leqslant_{\mathbb{L}} y \Leftrightarrow x \leqslant_{\mathbb{P}} gy. \tag{1.3}$$

In this context, we call f **residuated** and g **residual**.

Definition 1.11 (Galois connection)**.** Let $\mathbb{P} := (P, \leqslant)$ and $\mathbb{L} := (L, \leqslant)$ be posets and $f : P \to L$, $g : L \to P$ maps. The pair (f, g) is a **Galois connection** w.r.t. $(\mathbb{P}, \mathbb{L})$ if (f, g) is an adjunction w.r.t. $(\mathbb{P}, \mathbb{L}^{\mathrm{d}})$.

In the following we are going to present some theorems, that help establish the basic concept of an adjunction.

Theorem 1.1. Let $\mathbb{P} = (P, \leqslant_{\mathbb{P}})$, $\mathbb{L} = (L, \leqslant_{\mathbb{L}})$ be ordered sets and let $(\mathbb{P}, \mathbb{L}, f, g)$ be a poset adjunction then f is isotone.

Proof. Let $x \leqslant x'$. Because (f, g) is an adjunction w.r.t. $(\mathbb{P}, \mathbb{L})$, the following holds:

$$fx \leqslant fx \Leftrightarrow x \leqslant g(fx)$$
$$fx' \leqslant fx' \Leftrightarrow x' \leqslant g(fx').$$

$x \leqslant x'$ leads to:

$$fx \leqslant fx' \Leftrightarrow x \leqslant g(fx').$$

□

In an adjunction f and g are clearly defined, as the next theorem shows:

Theorem 1.2. Let (f,g) be an adjunction w.r.t. $(\mathbb{P},\mathbb{L})$ and (f,h) be an adjunction w. r. t. $(\mathbb{P},\mathbb{L})$. It holds that $g = h$.

Proof. Let (f,g) and (f,h) are adjunctions, which means that, for $x \in \mathbb{P}$ and $y \in \mathbb{L}$ the following holds:

$$fx \leqslant y \Leftrightarrow x \leqslant gy \tag{$*$}$$
$$fx \leqslant y \Leftrightarrow x \leqslant hy. \tag{$**$}$$

Because $gy \leqslant gy$ it applies:

$$gy \leqslant gy \overset{(*)}{\Leftrightarrow} f(gy) \leqslant y \overset{(**)}{\Leftrightarrow} gy \leqslant hy.$$

But also for $hy \leqslant hy$ the following holds:

$$hy \leqslant hy \overset{(**)}{\Leftrightarrow} f(hy) \leqslant y \overset{(*)}{\Leftrightarrow} hy \leqslant gy$$

and thus $g = h$.

□

The next theorem points out that the concatenation of adjunctions leads to an adjunction again.

Theorem 1.3. Let (f_1,g_1) be an adjunction w.r.t. $(\mathbb{P},\mathbb{M})$ and (f_2,g_2) an adjunction w.r.t. $(\mathbb{M},\mathbb{L})$, then (f,g) build an adjunction w.r.t. $(\mathbb{P},\mathbb{L})$, in which $f := f_2 \circ f_1$ and $g := g_1 \circ g_2$.

Proof. Let $x \in P$ and $y \in L$ with

$$fx = f_2(f_1 x) \leqslant_{\mathbb{M}} y.$$

Because (f_2,g_2) is an adjunction, it applies:

$$f_1 x \leqslant_{\mathbb{L}} g_2 y,$$

and the adjunction (f_1, g_1) leads to:

$$x \leqslant_{\mathbb{P}} g_1(g_2 y) = gy$$

and finally:

$$fx \leqslant y \Leftrightarrow x \leqslant gy.$$

□

The duality principle leads to the following theorem:

Theorem 1.4. Let (f,g) be an adjunction w.r.t. $(\mathbb{P},\mathbb{L})$, then (g,f) is an adjunction w.r.t. $(\mathbb{L}^d,\mathbb{P}^d)$.

Proof. (f,g) is an adjunction, so for all $x \in P, y \in L$ the following holds:

$$\begin{aligned} & fx \leqslant y \Leftrightarrow x \leqslant gy \\ \Leftrightarrow\ & (x \leqslant gy \Leftrightarrow fx \leqslant y) \\ \Leftrightarrow\ & (gy \leqslant^d x \Leftrightarrow y \leqslant^d fx). \end{aligned}$$

□

The next theorem shows the connection between kernel/closure operators and adjunctions. The theorem can be found on page 17 in [Pig10].

Theorem 1.5. Let (f,g) be an adjunction w.r.t. $(\mathbb{P},\mathbb{L})$, $\mathbb{P} := (P,\leqslant)$ and $\mathbb{L} := (L,\leqslant)$ partially ordered sets, then $h := g \circ f$ is a closure operator on $\mathbb{P}$ and $k := f \circ g$ a kernel operator on $\mathbb{L}$.

Proof. We show that h fulfills the Equivalence 1.2.

"⇒ ": Let $x, y \in P$ with $x \leqslant g(fy)$. Because (f,g) is an adjunction and Equivalence 1.3 holds:

$$fx \leqslant fy \qquad (*)$$

and therefore:

$$\begin{aligned} & g(fx) \leqslant g(fx) \\ \overset{2.3}{\Rightarrow} & f(g(fx)) \leqslant f(x) \overset{*}{\leqslant} fy \\ \Rightarrow & f(g(fx)) \leqslant fy \\ \overset{2.3}{\Rightarrow} & hx = g(fx) \leqslant g(fy) = hy. \end{aligned}$$

"$\Leftarrow$": Let $g \circ fx \leqslant g \circ fy$ hold.

$$\begin{aligned} & f(x) \leqslant f(x) \\ \overset{2.3}{\Rightarrow} & x \leqslant g(fx) \\ \Rightarrow & x \leqslant g(fy). \end{aligned}$$

The duality principle provides that k is a kernel operator.

□

Lemma 1.2. Let $\mathbb{P} := (P, \leqslant)$ be a poset and $T \subseteq P$, then T is a kernel system in $\mathbb{P}$ if the canonical embedding τ of $\mathbb{P}|T$ into $\mathbb{P}$ is residuated. Dually, T is a closure system in $\mathbb{P}$ if the cannonical embedding τ of $\mathbb{P}|T$ into $\mathbb{P}$ is residual.

Proof. We assume τ is residuated, so there exists a map ϕ, such that (τ, ϕ) build an adjunction w.r.t. $(\mathbb{P}, \mathbb{P}|T)$. Theorem 1.5 shows that $\tau \circ \phi$ is a kernel operator and the one-to-one correspondence described in Lemma 1.1 provides that T is a kernel system. The other statement follows dually.

□

Theorem 1.6. Let (f, g) be an adjunction w.r.t. $(\mathbb{P}, \mathbb{L})$, then the following holds:

$$f \circ g \circ f = f \text{ and } g \circ f \circ g = g.$$

Proof. Let **id** be the identity map. Then $g \circ f \geqslant \mathbf{id}$ because $g \circ f$ is a closure operator and $f \circ g \leqslant \mathbf{id}$ in view of the fact that $f \circ g$ is a kernel operator and therefore:

$$g = g \circ \mathbf{id} \leqslant g \circ f \circ g$$

but also:

$$g = \mathbf{id} \circ g \geqslant g \circ f \circ g$$

which leads directly to $g = g \circ f \circ g$. The second proposition follows dually.

□

The next theorem helps to characterize residuated maps.

Theorem 1.7. Let $\mathbb{P} := (P, \leqslant)$ and $\mathbb{L} := (L, \leqslant)$ be complete lattices and $f : P \to L$ a supremum preserving map. This means, for every subset X of P the equation:

$$f(\sup X) = \sup f(X)$$

holds. Then f, together with the map

$$g : L \to P,\ y \mapsto \sup\{t \in P \mid ft \leqslant y\},$$

forms an adjunction w.r.t. $(\mathbb{P}, \mathbb{L})$.

Proof.
" $\Rightarrow$ ": Let $x \in P$ and $y \in L$ with $fx \leqslant y$. That yields to $x \leqslant gy$ because of the definition of g.
" $\Leftarrow$ ": Let $x \leqslant gy$. Then the following holds:

$$\begin{aligned} f(gy) &= f(\sup\{t \mid t \in P : ft \leqslant y\}) \\ &= \sup\{ft \mid t \in P : ft \leqslant y\} \\ &\leqslant y. \end{aligned}$$

□

The above theorem shows that supremum preserving maps are residuated. However, residuated maps are supremum preserving too as the following theorem shows.

Theorem 1.8. Let $\mathbb{P} := (P, \leqslant)$ and $\mathbb{L} := (L, \leqslant)$ be posets and (f, g) an adjunction w.r.t. $(\mathbb{P}, \mathbb{L})$. Then f preserves all existing suprema.

Proof. Let $X \subseteq P$ and $s = \sup X$ exist.
Then, $fx \leqslant fs$ for all $x \in X$, which means fs is upper bound of fX because $x \leqslant s$ for all $x \in X$ and f is an isotone map, as shown in Theorem 1.1. We have to prove that fs is the least upper bound. Therefore, let t be an upper bound of fX in $\mathbb{L}$. That follows to $fx \leqslant t$ for all $x \in X$. Because (f, g) is an adjunction, we get $x \leqslant gt$ for all $x \in X$. So, gt is upper bound of X in $\mathbb{P}$. In our case, s is the supremum and, therefore, the least upper bound of X in $\mathbb{P}$, and so it follows that $s \leqslant gt$, and hence (f, g) is an adjunction $fs \leqslant t$.

□

In the previous two theorems, we proved that every supremum preserving map is residuated and every residuated map is supremum preserving.

Lemma 1.3. Let $\mathbb{P} = (P, \leqslant)$ and $\mathbb{L} = (L, \leqslant)$ be posets and $f : P \to L$ a residuated map, then the preimage of a principal ideal in $\mathbb{L}$ under f is always a principal ideal in $\mathbb{P}$, that is, for every $y \in L$, there exists $x \in P$ such that

$$f^{-1}\{t \in L \mid t \leqslant y\} = \{s \in P \mid s \leqslant x\}.$$

However, a map $g : L \to P$ is residual if the preimage of a principal filter in $\mathbb{P}$ under g is always a principal filter in $\mathbb{L}$, that is, for every $x \in P$ there exists $y \in L$ s.t.

$$g^{-1}\{s \in P \mid x \leqslant s\} = \{t \in L \mid y \leqslant t\}.$$

Proof. This follows because f is isotone (Theorem 1.1) and preserves all existing suprema (Theorem 1.7). The second statement follows dually.

□

1.2 Foundations of Formal Concept Analysis

As we are going to see in this work, the concept of an adjunction is related to formal concept analysis. Therefore, we assembled some basic information of formal concept analysis, which can be found in [GK01].

Definition 1.12 (formal context)**.** A **formal context** $\mathbf{K} := (G, M, I)$ consists of two sets G and M and a binary relation I between G and M. The elements of G are called **objects** , the elements of M are called **attributes** of the context and $(g, m) \in I$ is read: the object g has the attribute m.

Example 1.1. The example shows the days from Monday to Friday as objects. Attributes are sunny, cloudy and rainy to describe the weather on a particular day.

	sunny	cloudy	rainy
Monday	×		
Tuesday		×	
Wednesday		×	×
Thursday		×	
Friday	×		

For reasons of clarity and comprehensibility, formal contexts are commonly represented as cross tables, as can be seen in Example 1.1.

Definition 1.13 (derivation operator)**.** Let (G,M,I) be a formal context. For a subset $A \subseteq G$ of objects we define

$$A' := \{m \in M \mid gIm \text{ for all } g \in A\}$$

(the set of common attributes of all objects in A). For the set of objects {Monday,Friday} from the example above we get the following derivation:

$$\{\text{Monday, Friday}\}' := \{\text{sunny}\}$$

For a set of attributes $B \subseteq M$ the derivation operator is defined as:

$$B' := \{g \in G \mid gIm \text{ for all } m \in B\}.$$

It is the set of objects which have all atributes of B. From the above example:

$$\{\text{cloudy}\}' := \{\text{Tuesday, Wednesday, Thursday}\}.$$

Definition 1.14 (Dedekind-MacNeille completion)**.** Let $\mathbb{D} := (D, \leqslant)$ be an ordered set. For every subset $A \subseteq D$ we call

$$A^{\uparrow} := \{d \in D \mid a \leqslant d \text{ for all } a \in A\}$$

the set of **all upper bounds** of A. The set of **all lower bounds** of A is defined dually and denoted by $A_{\downarrow}$. A **cut** of $\mathbb{D}$ is a pair (A,B) with $A,B \subseteq D$, $A^{\uparrow} = B$ and $A = B_{\downarrow}$. It is well known that these cuts ordered by

$$(A_1,B_1) \leqslant (A_2,B_2) :\Leftrightarrow A_1 \subseteq A_2 \ (\Leftrightarrow B_2 \subseteq B_1),$$

form a complete lattice, the **Dedekind-MacNeille completion**. It is the smallest complete lattice containing a subset that is order isomorphic with $\mathbb{D}$ and denoted by $DMN(\mathbb{D})$.

Now we come to two of the most important definitions of this thesis:

Definition 1.15 (pattern setup)**.** A triple $\mathcal{P} = (G, \mathbb{D}, \delta)$ is a **pattern setup** if G is a set of so-called **objects**, $\mathbb{D} = (D, \sqsubseteq)$ is a poset of so-called **patterns**, and $\delta : G \to D$ is a map.

Assuming slightly stricter requirements for the set of patterns, we receive a pattern structure, as the following definition shows.

Definition 1.16 (pattern structure)**.** A **pattern structure** is defined as a triple $(G, \mathbb{D}, \delta)$ where G is a set of so-called **objects**,

$$\mathbb{D} := (D, \sqsubseteq)$$

forms a meet-semilattice of so-called **patterns**, and

$$\delta : G \longrightarrow D$$

is a map, such that every subset X of

$$\delta G := \{\delta g \mid g \in G\}$$

has an infimum (greatest lower bound) in $\mathbb{D}$, denoted by $\bigsqcap X$. The set D_δ of all infima of subsets of δG forms a complete subsemilattice of $\mathbb{D}$.

The following definition will be needed in the next chapter.

Definition 1.17 (projection)**.** Let $(G, \mathbb{D}, \delta)$ be a pattern structure, then a kernel operator ψ on $\mathbb{D}$ will be called a **projection**.

1.3 Clustering Algorithms

In the application part of this thesis clustering algorithms are going to become important. As shown, for example, in [VM02], [SQT02], [CH74], [KS96] and [ESBB98], clustering algorithms have various fields of application. We present two of the most important clustering algorithms in this section. We start with the well known k-means algorithm.

1.3.1 The k-means Algorithm

This section refers to [AV06]. The **k-means algorithm** is a widely used clustering technique that seeks to minimize the average squared distance between points in the same cluster. Although it offers no accuracy guarantees, its simplicity and speed are very appealing in practice. For the *k-means* problem, we are given an integer k and a set of n data points $X \subset \mathbb{R}^n$. We wish to choose k centers C, so as to minimize the potential function,

$$\Theta = \sum_{x \in X} \min_{c \in C} ||x - c||^2.$$

The *k-means* algorithm follows these steps:

1. Arbitrarily choose an initial k centers $C = \{c_1, c_2, ..., c_k\}$.
2. For each $i \in \{1, ..., k\}$ set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$.
3. For each $i \in \{1, ..., k\}$ set c_i to be the center of mass of all points in $C_i : c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.
3. Repeat step 2 and 3 until C no longer changes.

Let $D(x)$ denote the shortest distance from a data point to the closest center we have already chosen. Then, the following improvement of the algorithm can be made:

1a. Take one center c_1, chosen uniformly at random from X.

1b. Take a new center c_i, choosing $x \in X$ with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$.

1c. Repeat step 1b. until we have taken k centers altogether.

2-4. Proceed as with the standard *k-means* algorithm.

1.3.2 The k-mediods Algorithm

Next, we present the well known **k-medoids algorithm** introduced in [PJ09]. Let V be a set and $k \in \mathbb{N}$, then the k-mediods algorithm $med(V,k)$ works as follows:

1. (Select initial medoids)

   1-1. Calculate the distance between every pair of all objects based on the chosen dissimilarity measure.

   1-2. Calculate v_j for object j as follows:

$$v_j = \sum_{i=1}^{n} \frac{d_{ij}}{\sum_{l=1}^{n} d_{il}}, \ j = 1, ..., n$$

   1-3. Sort v_j's in ascending order. Select k objects having the first k smallest values as initial mediods.

   1-4. Obtain the initial cluster result by assigning each object to the nearest mediod.

   1-5. Calculate the sum of distances from all objects to their medoids.

2. (Update medoids)

   2-1. Find a new medoid of each cluster, which is the object minimizing the total distance to other objects in the cluster by replacing it with the new medoid.

3. (Assign objects to medoids)

3-1. Assign each object to the nearest medoid and obtain the cluster result.

3-2. Calculate the sum of distances from all objects to their medoids. If the sum is equal to the previous one, stop the algorithm. Otherwise, go back to 2.

The clustering algorithms introduced here are the most important for this thesis. For a nice overview of clustering algorithms, we recommend [Mad12]. Theoretically, all clustering algorithms presented in [Mad12] can be used in our application chapter. The clustering algorithm introduced here are those we considered most relevant for this thesis.

2

Examples of Pattern Structures

In this section we are going to breath life into the definitions of pattern structures by presenting some examples. Parts of this section have been published in [LS20a].

2.1 Elementary Pattern Structures

First, we give an example of a pattern structure. Then, we introduce a general definition to create a pattern structure through a formal context.

Example 2.1. Our starting point is the formal context $\mathbf{K} := (G, M, I)$ of Example 1.1, with

$$G := \{\text{Monday, Tuesday, Wednesday, Thursday, Friday}\}$$

and as the set of patterns $\mathbb{D}$ we use the dually ordered powerset $\mathbf{2}^M := (2^M, \supseteq)$ of the set of atributes

$$M := \{\text{sunny, cloudy, rainy}\}.$$

The pattern set D is visualized in the following figure.

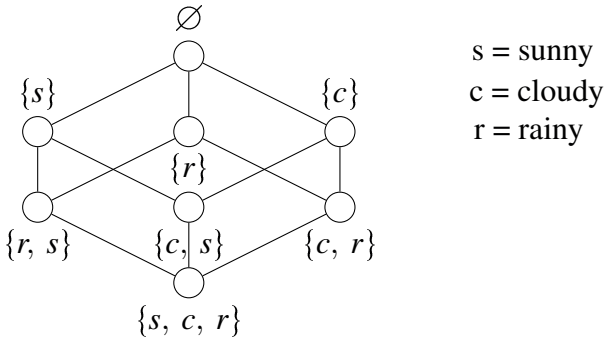


Figure 2.1: dually ordered powerset lattice $\mathbb{D} := \mathbf{2}^M$ with $D := 2^M$

The map

$$\delta : G \to 2^M, g \mapsto \{m \mid m \in M, gIm\}$$

assigns a pattern to every subset of objects and, since every powerset is a complete lattice,

$$\mathcal{P} := (G, \mathbf{2}^M, \delta)$$

forms a pattern structure.

The example gives rise to a more general definition of an elementary pattern structure.

Definition 2.1 (elementary pattern structure)**.** Let $\mathcal{P} := (G, \mathbb{D}, \delta)$ be a pattern setup, then the **elementary pattern structure** is given by:

$$(G, \mathbb{E}, \varepsilon) \text{ with } \mathbb{E} := (2^D, \supseteq) \text{ and } \varepsilon : G \to 2^D, g \mapsto \{\delta g\}.$$

This definition gives us the opportunity to create a pattern structure out of a given pattern setup. Next, we want to use this definition to show a connection between formal contexts and pattern structures.

Definition 2.2 (associated pattern structure)**.** Let $\mathbf{K} := (G, M, I)$ be a formal context. Then

$$\mathcal{P} := (G, \mathbf{2}^M, \delta) \text{ with}$$
$$\delta : G \to 2^M, g \mapsto \{m \mid m \in M, gIm\}$$

forms a pattern structure. It is called the **associated pattern structure** of $\mathbf{K}$.

The elementary pattern structure reveals how to build a pattern structure out of a given pattern setup. Below we present further investigations of the relationship between pattern setups and pattern structures.

2.2 Embedded Pattern Structures

In this section, we present a toolbox to describe the connection between pattern setups and pattern structures. We start with a definition.

Definition 2.3 (embedded pattern structure)**.** Let $\mathbb{L}$ be a complete lattice and $\mathcal{P} := (G, \mathbb{D}, \delta)$ a pattern setup with $\mathbb{D} := \mathbb{L}|D$. Then we call

$$\mathcal{P}_e := (G, \mathbb{L}, \mathbb{D}, \overline{\delta}) \text{ with } \overline{\delta} : G \to L, g \mapsto \delta g$$

the **embedded pattern structure**. We say the pattern setup $\mathcal{P}$ is **embedded** in the pattern structure $(G, \mathbb{L}, \overline{\delta})$.

The following two constructions are a demonstration of how to build an embedded pattern structure from a pattern setup.

Construction 2.1. Let G be a set and let $\mathbb{D} := (D, \leqslant)$ be a poset. Then for every map $\rho : G \to D$ the elementary pattern structure is given by:

$$\mathcal{P}_\rho := (G, \mathbb{L}, \varepsilon) \text{ with } \mathbb{L} := (2^D, \supseteq) \text{ and } \varepsilon : G \to 2^D, g \mapsto \{\rho g\}.$$

Hence, the pattern setup $(G, \mathbb{D}, \rho)$ is embedded in the pattern structure $\mathcal{P}_\rho$.

Next we use Definition 1.14 to build a pattern structure from a given pattern setup.

Construction 2.2. For every pattern setup $\mathcal{P} := (G, \mathbb{D}, \delta)$, a pattern structure is given through the Dedekind-MacNeille completion and

$$(G, DMN(\mathbb{D}), \overline{\delta}) \text{ with } \overline{\delta} : G \to DMN(\mathbb{D}), g \mapsto \delta g.$$

The two constructions point out that every pattern setup can be embedded in a pattern structure. This is a useful piece of information for many real world applications where only a pattern setup is given.

2.3 Interval Pattern Structures

In this last part of the chapter we introduce another pattern structure, which is very important for the application part of this thesis. As shown for example in [KKND11] we can construct a pattern structure from intervals. To build such a pattern structure on a set we need the following definition:

Definition 2.4 (interval poset)**.** Let $\mathbb{P} := (P, \leqslant_{\mathbb{P}})$ be a poset. Then we call

$$\mathrm{Int}\mathbb{P} := \{[p,q]_{\mathbb{P}} \mid p,q \in \mathbb{P}\} \text{ with } [p,q]_{\mathbb{P}} := \{t \in P \mid p \leqslant_{\mathbb{P}} t \leqslant_{\mathbb{P}} q\}$$

the **set of all intervals** of $\mathbb{P}$ and

$$\mathbb{I}\mathrm{nt}\mathbb{P} := (\mathrm{Int}\mathbb{P}, \supseteq)$$

the **interval poset** of $\mathbb{P}$. Furthermore, if $\mathbb{P}$ is a lattice we define an intersection operator $\sqcap$ on $\mathrm{Int}\mathbb{P}$ in the following way:

$$[p_1,q_1] \sqcap [p_2,q_2] = [p_1 \wedge p_2, q_1 \vee q_2] \text{ for all } [p_1,q_1],\ [p_2,q_2] \in \mathrm{Int}\mathbb{P}.$$

Remarks:

(1) Let $\mathbb{P}$ be a poset. Then $\mathbb{I}\mathrm{nt}\mathbb{P}$ is an upper bounded poset, that is, $\varnothing$ is the greatest element of $\mathbb{I}\mathrm{nt}\mathbb{P}$, provided $\mathbb{P}$ has at least 2 elements. Furthermore, if $\mathbb{P}$ is a (complete) lattice then so is $\mathbb{I}\mathrm{nt}\mathbb{P}$.

(2) If A is a set of attributes then $(\mathbb{R}^A, \leqslant) := (\mathbb{R}, \leqslant)^A$ is a lattice. With (1) it follows that $\mathbb{I}\mathrm{nt}(\mathbb{R}^A, \leqslant)$ is an upper bounded lattice.

Definition 2.5 (evaluation map, evaluation setup)**.** Let G be a finite set, M a set of attributes and $\mathbb{W}_m := (W_m, \leqslant_m)$ a complete lattice for every attribute $m \in M$. Further, let

$$W := \prod_{m \in M} W_m \text{ and } \mathbb{W} := \prod_{m \in M} \mathbb{W}_m.$$

Then, a map

$$\alpha : G \to W, g \mapsto \prod_{m \in M} \{\alpha_m g\}$$

such that

$$\alpha_m : G \to W_m, g \mapsto w_m$$

is called **evaluation map**. We call $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ an **evaluation setup**.

Theorem 2.1. Let G be a finite set, A a set of attributes and α an evaluation map. Then

$$\mathcal{P}_\alpha := (G, \mathbb{I}\mathrm{nt}(\mathbb{R}^A, \leqslant), \delta)$$

with

$$\delta : G \to \mathrm{Int}(\mathbb{R}^A, \leqslant),\ g \mapsto [\alpha g, \alpha g] = \{\alpha g\}$$

is a pattern structure.

Remarks:

(1) For the above, it is necessary to include the empty set as an empty interval to assure the existence of the infimum of the empty map into the (dually ordered) interval lattice.

(2) We illustrate the above in the following example:

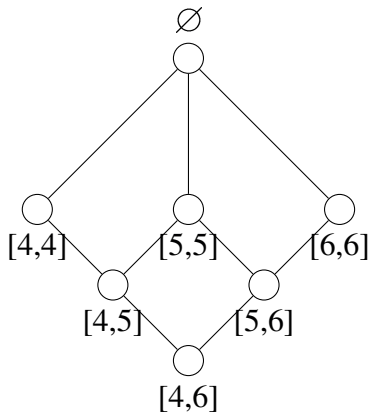


Figure 2.2: Meet-semi-lattice $(D, \sqcap)$ with $D := \{[4,4], [5,5], [6,6], [4,5], [5,6], [4,6], \varnothing\}$

In this chapter we introduced a general framework of interval pattern structures.The application part of this thesis contains concrete examples of the presented framework.

Remark: There is a connection between interval pattern structures and fuzzy formal contexts as for example shown in [PK12]. Fuzzy contexts are not treated in this thesis but because of their high importance of them we mentioned it here. For an itroduction to fuzzy FCA we recommend [Bel12, Bel11, JFG94].

3
A Counterexample

Pattern structures within the framework of formal concept analysis were introduced in [GK01]. Since then, they have turned out to be a useful tool for analysing various real-world applications (cf. [GK01, Kuz09, Kuz13, KKND11, KS11]). In this chapter, we will point out that the theoretical foundations of pattern structures encourage still some fruitful discussions. In particular, the role projections play within pattern structures for information reduction still needs some further investigation.
The goal of this chapter is to determine the circumstances under which pattern structures can or cannot be replaced by simpler (meaningful) ones. Here it turns out that projections do not always give rise to new pattern structures, as the literature (for example [GK01, Kuz09]) suggests. However, residual projections, which are also introduced in this part of the thesis, do. This chapter has been published in [LS15a].

3.1 Preliminaries

In the following, we construct a counterexample of a projection, that does not give rise to a pattern structure. We need the following preparations:

Definition 3.1 (vertical sum)**.** Let $\mathbb{P}_1 = (P_1, R_1)$ and $\mathbb{P}_2 = (P_2, R_2)$ be posets with $P_1 \cap P_2 = \varnothing$. Then the **vertical sum** of $\mathbb{P}_1$ with $\mathbb{P}_2$ is defined as $\mathbb{P} := (P, R)$ with $P := P_1 \cup P_2$ and

$$R := R_1 \cup R_2 \cup (P_1 \times P_2);$$

we set

$$\mathbb{P}_1 +_{\mathit{vert}} \mathbb{P}_2 := \mathbb{P}.$$

If $(G, \mathbb{D}, \delta)$ is a pattern structure, then a kernel operator ψ on $\mathbb{D}$ will also be called a projection. Literature, compare for example [GK01, Kuz09], suggests that any such projection ψ on a pattern structure $(G, \mathbb{D}, \delta)$ induces a pattern structure via $(G, \mathbb{D}, \psi \circ \delta)$. However, the following example shows that this is not always the case.

3.2 Construction of the Counterexample

Consider the chain $\mathbb{P}_0 := (\mathbb{N}, \leqslant)$ and the complete chain $\mathbb{P} := (\bar{\mathbb{N}}, \leqslant)$, where $\bar{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$ and $x \leqslant \infty$ for all $x \in \bar{\mathbb{N}}$; then

$$\mathbb{D} := \mathbb{P}_0 \; +_{\mathit{vert}} \; (\mathbb{P}^d \times \mathbb{P}_0^d)$$

is a lattice visualised in the following figure:

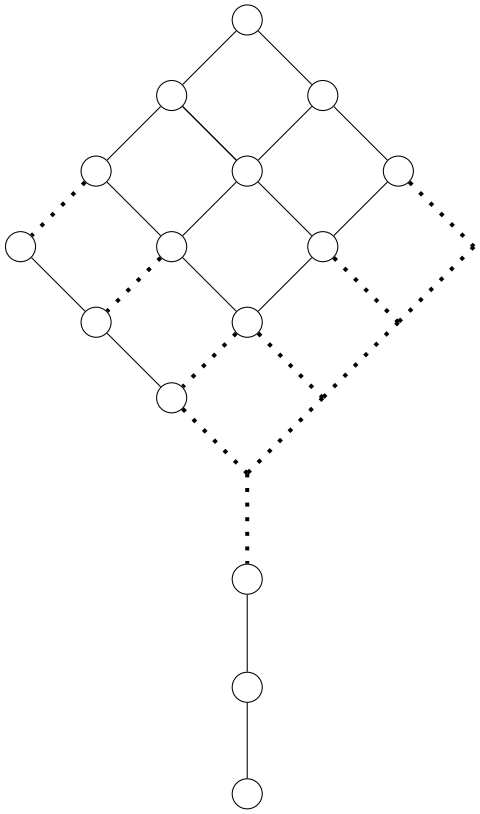


Figure 3.1: Visualisation of $\mathbb{D}$

For $G := \mathbb{N}$, the map

$$\delta : G \longrightarrow D, n \mapsto (n, 0)$$

gives rise to a pattern structure $(G, \mathbb{D}, \delta)$, where $D_\delta = \bar{\mathbb{N}} \times \{0\}$. For

$$\Delta_{\mathbb{N}} := \{(n, n) \mid n \in \mathbb{N}\}$$

the set

$$K := \Delta_{\mathbb{N}} \cup \{0\}$$

forms a kernel system. The associated kernel operator ψ has the property that $(G, \mathbb{D}, \psi \circ \delta)$ is ***not*** a pattern structure because $\Delta_{\mathbb{N}}$ has no infimum in $\mathbb{D}$ (see also Figure 3.2).

This may be even more surprising since ψ preserves finite meets in $\mathbb{D}$ and K forms a sublattice of $\mathbb{D}$.

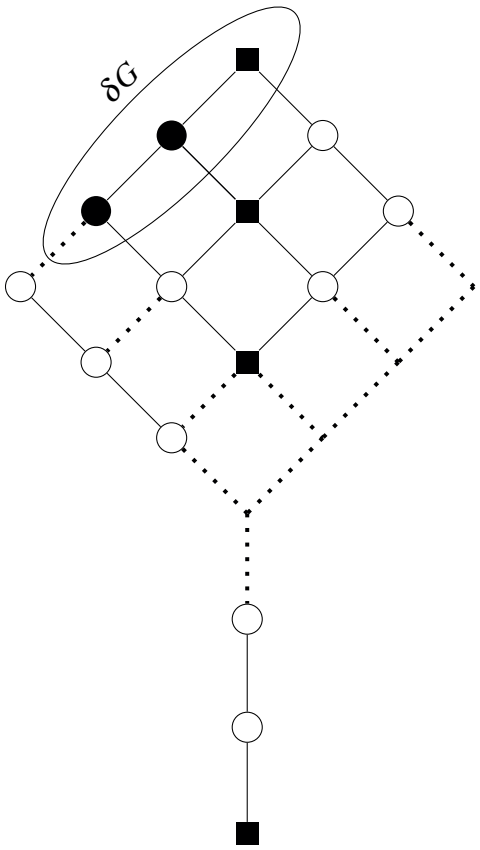


Figure 3.2: Visualisation of the counterexample

3.3 Bipolar Systems

We are going to show that every residual projection on a pattern structure induces a pattern structure. For this we need to prepare the following:

Definition 3.2 (bipolar system)**.** A **bipolar system** in a poset $\mathbb{P}$ is defined as a kernel system in $\mathbb{P}$ which is also a closure system in $\mathbb{P}$. The set of all bipolar systems in $\mathbb{P}$ will be denoted by $BS\mathbb{P}$.

Lemma 3.1. For a poset $\mathbb{P} = (P, \leqslant)$, a map $\psi : P \to P$ is a kernel operator on $\mathbb{P}$ if and only if the following holds for all $x, y \in P$:

$$x \leqslant \psi y \Leftrightarrow \exists\, t \in \psi P(x \leqslant t \leqslant y). \qquad (*)$$

Dually, a map $\varphi : P \to P$ is a closure operator on $\mathbb{P}$ if and only if the following holds for all $x, y \in P$:

$$\varphi x \leqslant y \Leftrightarrow \exists\, t \in \varphi P(x \leqslant t \leqslant y). \qquad (**)$$

Proof. First, assume that ψ is a kernel operator on $\mathbb{P}$. Then $x \leqslant \psi y$ implies $x \leqslant t \leqslant y$ for $t := \psi y$ (since ψ is contractive, that is, $t \leqslant y$). On the other hand, if $x \leqslant t \leqslant y$ holds for $t \in \psi P$ then $t \leqslant \psi y$ (since ψ is a kernel operator), which immediately implies $x \leqslant \psi y$.
Secondly, assume that $(*)$ holds. Then $\psi y \leqslant \psi y$ implies $\psi y \leqslant t \leqslant y$ for some $t \in \psi P$; therefore, $\psi y \leqslant y$ holds. That is, ψ is contractive. Also, for $t \in \psi P$ with $t \leqslant y$ we get $t \leqslant t \leqslant y$, which by $(*)$ implies $t \leqslant \psi y$. It follows that ψ is a kernel operator on $\mathbb{P}$.

□

In the following theorem we will clarify the connection between residual kernel operators and so-called bipolar systems in posets.

Theorem 3.1. If $\mathbb{P}$ is a poset, $KO^r\mathbb{P}$ denote the set of all residual kernel operators on $\mathbb{P}$, and dually, let $CO_r\mathbb{P}$ denote the set of all residuated closure operators on $\mathbb{P}$, then the map

$$KO^r\mathbb{P} \to BS\mathbb{P},\ \psi \mapsto \psi P$$

is a bijection, and dually, the map

$$CO_r\mathbb{P} \to BS\mathbb{P},\ \psi \mapsto \psi P$$

is a bijection too.

Proof. Since

$$KO\mathbb{P} \to KS\mathbb{P}, \psi \mapsto \psi P$$

is a bijection (see Lemma 1.1), we have to show that $\psi \in KO^r\mathbb{P}$ always implies that ψP is a bipolar system in $\mathbb{P}$. On the other hand, we have to verify that the kernel operator associated

with a bipolar system in $\mathbb{P}$ is always residual.
First, let $\psi \in KO^r\mathbb{P}$. Then ψP is a kernel system in $\mathbb{P}$. Since ψ is residual on $\mathbb{P}$, there exists φ such that (φ, ψ) is an adjunction on $\mathbb{P}$. For $x \in P$, the element $u := \psi(\varphi x) \in \psi P$ satisfies $x \leqslant u$ (since $\varphi x \leqslant \varphi x$ and the pair (φ, ψ) is an adjunction). Also, for $y \in P$, the element $w := \psi y \in \psi P$ with $x \leqslant w$ satisfies $\varphi x \leqslant y$. Thus $u \leqslant w$ (since ψ is isotone on $\mathbb{P}$), which yields that u is the least element in ψP with $x \leqslant u$. It follows that ψP is also a closure system and therefore a bipolar system in $\mathbb{P}$.
Second, let $B \in BS\mathbb{P}$. The associated kernel operator of B in $\mathbb{P}$ will be denoted by ψ and the associated closure operator of B in $\mathbb{P}$ will be denoted by φ. Since $\varphi P = B = \psi P$, the conditions $(**)$ and $(*)$ of Lemma 3.1 yield the following for all $x, y \in P$:

$$\varphi x \leqslant y \Leftrightarrow \exists\, t \in B\,(x \leqslant t \leqslant y) \Leftrightarrow x \leqslant \psi y.$$

Therefore, (φ, ψ) is an adjunction on $\mathbb{P}$, from which we derive that φ is a residuated closure operator and ψ is a residual kernel operator with $\varphi P = B = \psi P$.

□

3.4 Residual Projections

Our final result will state that residual projections on pattern structures induce pattern structures again. We start with a definition.

Theorem 3.2. Let $\mathbb{S} := (G, \mathbb{D}, \delta)$ be a pattern structure with $\mathbb{D} := (D, \sqsubseteq)$, and let ψ be a **residual projection** on $\mathbb{S}$, that is, ψ be a residual kernel operator on $\mathbb{D}$.
Then,

$$\mathbb{S}_\psi := (G, \mathbb{D}, \psi \circ \delta)$$

is a pattern structure, which satisfies $D_{\psi\circ\delta} = \psi(D_\delta)$. Furthermore, ψD is a bipolar system in $\mathbb{D}$ satisfying that

$$(G, \mathbb{D} \,|\, \psi D, \psi \circ \delta)$$

is a pattern structure.

Proof. Let Y be a subset of $\psi(\delta G)$. Then there exists a subset X of δG with $Y = \psi X$. Since $\mathbb{S}$ is a pattern structure, $\bigsqcap X$ exists in $\mathbb{D}$.
Claim: The infimum of Y in $\mathbb{D}$ is given by $\psi(\bigsqcap X)$, thus $\psi(\bigsqcap X) = \bigsqcap Y$.
Since $\bigsqcap X$ is a lower bound of X in $\mathbb{D}$, the element $\psi(\bigsqcap X)$ is a lower bound of $Y = \psi X$ in $\mathbb{D}$ (because ψ is isotone). Let now t' be a lower bound of Y in $\mathbb{D}$. Based on our assumption on ψ, there exists φ such that (φ, ψ) is an adjunction on $\mathbb{P}$. Thus, for every $x \in X$, we have $t' \leqslant \psi x$ and therefore $\varphi t' \leqslant x$, which means that $\varphi t'$ is a lower bound of X. This implies $\varphi t' \leqslant \bigsqcap X$, and this yields $t' \leqslant \psi(\bigsqcap X)$.
We conclude that $\mathbb{S}_\psi$ is a pattern structure. This, together with Theorem 3.1, completes the proof.

□

We have laid out that projections do not always give rise to new pattern structures. As a solution, we presented the concept of residual projections. In [BKN15b], o-projections are defined, which also lead to pattern structures again.

Definition 3.3 (o-projection)**.** Given a pattern structure $\mathcal{P} := (G, \mathbb{D}, \delta)$ with $\mathbb{D} := (D, \sqsubseteq)$ and a kernel operator ψ on D, the **o-projected pattern structure**, or short **o-projection** $\psi\mathcal{P}$ is a pattern structure

$$(G, \mathbb{D}_\psi, \psi \circ \delta) \text{ with } \mathbb{D}_\psi := (\psi D, \sqsubseteq_\psi),$$

where

$$\psi D = \{d \in D \mid \psi d = d\}$$

and

$$\forall x, y \in D,\ x \sqsubseteq_\psi y := \psi(x \sqsubseteq y).$$

Residual projections and o-projected pattern structures lead to pattern structures again. The next chapter is going to examine the relationship between them.

4

Pattern Structures and their Morphisms

Projections of pattern structures do not always lead to pattern structures. However, residual projections and o-projections do. As a unifying approach, we introduce the notion of pattern morphisms between pattern structures and provide a general sufficient condition for a homomorphic image of a pattern structure being a pattern structure again. In particular, we achieve a better understanding of the theory of o-projections.
The findings of this chapter have been published in [LS15b].

4.1 Introduction

The goal of this chapter is to establish an adequate concept of pattern morphism between pattern structures, which also gives a better understanding of the concept of o-projections as recently introduced and investigated in [BKN15b]. In Chapter 3, we showed that projections of pattern structures do not necessarily lead to pattern structures again. However, residual projections and o-projections do. It turns out that the concept of residual maps between the posets of patterns (with respect to two pattern structures) gives the key for a unifying view of o-projections and residual projections.
We also derived that a pattern morphism from a pattern structure to a pattern setup, which is surjective on the sets of objects, yields again a pattern structure.
Our main result states that a pattern morphism always induces an adjunction between the corresponding concept lattices. In case the underlying map between the sets of objects is surjective, the induced residuated map between the concept lattices turns out to be surjective too. This chapter is divided in two main parts. First we introduce a general framework. In the second part we apply this framework to the world of pattern structures and present some applications of our findings.

4.2 Adjunctions and their Concept Posets

Starting from a general framework, we require the following definitions.

Definition 4.1 (morphism)**.** Let

$$\mathcal{P} := (\mathbb{P}, \mathbb{S}, \sigma, \sigma^+) \text{ and}$$
$$\mathcal{Q} := (\mathbb{Q}, \mathbb{T}, \tau, \tau^+)$$

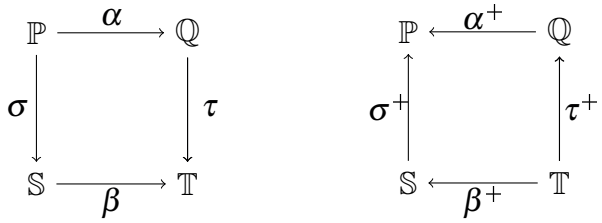
be poset adjunctions. Then, a pair (α, β) forms a **morphism** from $\mathcal{P}$ to $\mathcal{Q}$ if α^+ and β^+ exist, such that

$$(\mathbb{P}, \mathbb{Q}, \alpha, \alpha^+) \text{ and}$$
$$(\mathbb{S}, \mathbb{T}, \beta, \beta^+)$$

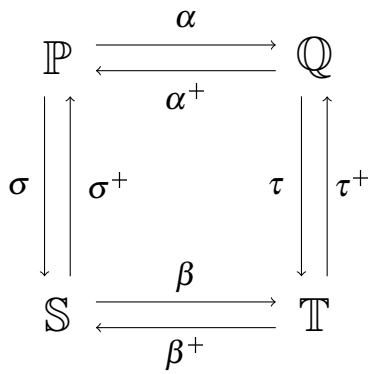
are poset adjunctions satisfying

$$\tau \circ \alpha = \beta \circ \sigma.$$

Remark: This implies $\alpha^+ \circ \tau^+ = \sigma^+ \circ \beta^+$, that is, the following diagrams are commutative:



Next we illustrate the involved poset adjunctions:



The following definition is about a relationship between poset adjunctions and formal concepts.

Definition 4.2 (concept poset)**.** For a poset adjunction $\mathcal{P} = (\mathbb{P}, \mathbb{S}, \sigma, \sigma^+)$ let

$$\mathrm{B}\mathcal{P} := \{(p,s) \in P \times S \mid \sigma p = s \wedge \sigma^+ s = p\}$$

denote the set of (**formal**) **concepts** in $\mathcal{P}$. Then the **concept poset** of $\mathcal{P}$ is given by

$$\mathbb{B}\mathcal{P} := (\mathbb{P} \times \mathbb{S}) \mid \mathrm{B}\mathcal{P},$$

that is, $(p_0, s_0) \leqslant (p_1, s_1)$ holds if and only if $p_0 \leqslant p_1$ and $s_0 \leqslant s_1$ for all $(p_0, s_0), (p_1, s_1) \in B\mathcal{P}$. If (p,s) is a formal concept in $\mathcal{P}$, then p is referred to as **extent** in $\mathcal{P}$ and s as **intent** in $\mathcal{P}$.

The next theorem links the above definitions.

Theorem 4.1. Let (α, β) be a morphism from a poset adjunction $\mathcal{P} = (\mathbb{P}, \mathbb{S}, \sigma, \sigma^+)$ to a poset adjunction $\mathcal{Q} = (\mathbb{Q}, \mathbb{T}, \tau, \tau^+)$. Then

$$(\mathbb{B}\mathcal{P}, \mathbb{B}\mathcal{Q}, \Phi_{(\alpha,\beta)}, \Phi^+_{(\alpha,\beta)})$$

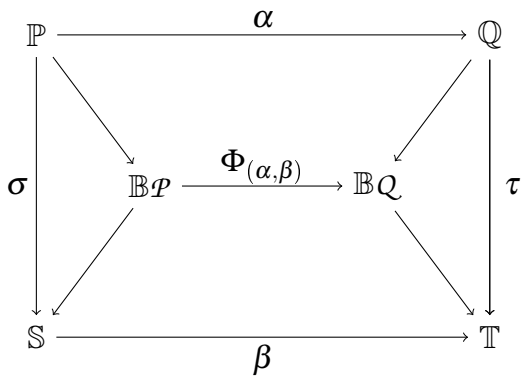
is a poset adjunction for

$$\Phi_{(\alpha,\beta)} : \mathrm{B}\mathcal{P} \to \mathrm{B}\mathcal{Q}, (p,s) \mapsto (\tau^+ \beta s, \beta s)$$

and

$$\Phi^+_{(\alpha,\beta)} : \mathrm{B}\mathcal{Q} \to \mathrm{B}\mathcal{P}, (q,t) \mapsto (\alpha^+ q, \sigma \alpha^+ q).$$

In addition, if α is surjective, then so is $\Phi_{(\alpha,\beta)}$.



Remark: In particular we want to point out that $\alpha^+ q$ is an extent in $\mathcal{P}$ for every extent q in $\mathcal{Q}$ and, similarly, βs is an intent in $\mathcal{Q}$ for every intent s in $\mathcal{P}$.

Proof. Let $(p,s) \in \mathrm{B}\mathcal{P}$ and $(q,t) \in \mathrm{B}\mathcal{Q}$; then

$$\begin{aligned}
&\sigma p = s \text{ and}\\
&\sigma^+ s = p \text{ and}\\
&\tau q = t \text{ and}\\
&\tau^+ t = q.
\end{aligned}$$

This implies $\beta s = \beta\sigma p = \tau\alpha p$, thus

$$\Phi_{(\alpha,\beta)}(p,s) = (\tau^+\beta s, \beta s) \in \mathrm{B}\mathcal{Q}$$

(since Theorem 1.6 $\tau\tau^+\beta s = \tau\tau^+\tau\alpha p = \tau\alpha p = \beta s$). Similarly, $\Phi^+_{(\alpha,\beta)}(q,t) \in \mathrm{B}\mathcal{P}$.

Assuming that $\Phi_{(\alpha,\beta)}(p,s) \leqslant (q,t)$ holds, implies $\beta s \leqslant t$. It follows that

$$\tau\alpha p = \beta\sigma p = \beta s \leqslant t$$

and hence with Theorem 1.3

$$p \leqslant \alpha^+\tau^+ t = \alpha^+ q,$$

that is, $(p,s) \leqslant \Phi^+_{(\alpha,\beta)}(q,t)$.

Conversely, assuming that $(p,s) \leqslant \Phi^+_{(\alpha,\beta)}(q,t)$ holds, implies $p \leqslant \alpha^+ q$. It follows that

$$p \leqslant \alpha^+ q = \alpha^+\tau^+ t = \sigma^+\beta^+ t,$$

and, hence, $\beta s = \beta\sigma p \leqslant t$, that is, $\Phi_{(\alpha,\beta)}(p,s) \leqslant (q,t)$.

Assume now that α is surjective; then $\alpha \circ \alpha^+ = \mathrm{id}_Q$. Let $(q,t) \in \mathrm{B}\mathcal{Q}$, that is, $\tau q = t$ and $\tau^+ t = q$. Then, for $p := \alpha^+ q$ and $s := \sigma p$, we have $(p,s) \in \mathrm{B}\mathcal{P}$ since

$$\sigma^+ s = \sigma^+\sigma\alpha^+ q = \sigma^+\sigma\alpha^+\tau^+ t = \sigma^+\sigma\sigma^+\beta^+ t = \sigma^+\beta^+ t = \alpha^+\tau^+ t = \alpha^+ q = p.$$

Our claim is now that $\Phi_{(\alpha,\beta)}(p,s) = (q,t)$ holds, that is, $\beta s = t$. The latter is true, since $\alpha p = \alpha\alpha^+ q = q$ implies

$$\beta s = \beta\sigma p = \tau\alpha p = \tau q = t.$$

□

Discussion for clarification: The question was raised whether, in the previous theorem, the residuated map $\Phi_{(\alpha,\beta)}$ from $\mathrm{B}\mathcal{P}$ to $\mathrm{B}\mathcal{Q}$ allows some modification, since the map

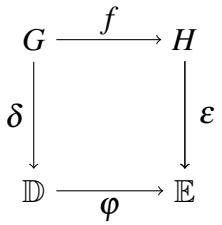
$$P \times S \to Q \times T, (p,s) \mapsto (\alpha p, \beta s)$$

is obviously residuated from $\mathbb{P} \times \mathbb{S}$ to $\mathbb{Q} \times \mathbb{T}$. However, in general, the latter map does not restrict to a map from $\mathrm{B}\mathcal{P}$ to $\mathrm{B}\mathcal{Q}$. Indeed, our construction of the map $\Phi_{(\alpha,\beta)}$ is of the form $(p,s) \mapsto (\alpha' p, \beta s)$. As a warning, we want to point out that, in general, there is no residuated map from $\mathrm{B}\mathcal{P}$ to $\mathrm{B}\mathcal{Q}$ of the form $(p,s) \mapsto (\alpha p, \beta' s)$. The simple reason for this is that βs is an intent in $\mathcal{Q}$ for every intent s in $\mathcal{P}$, while there may exist an extent p in $\mathcal{P}$ such that αp is not an extent in $\mathcal{Q}$.

4.3 Morphisms between Pattern Structures

In the following, we want to apply the theorem of the previous section to the world of pattern structures. We need the following preparation.

Definition 4.3 (pattern morphism)**.** If $\mathcal{G} = (G, \mathbb{D}, \delta)$ and $\mathcal{H} = (H, \mathbb{E}, \varepsilon)$ each is a pattern setup, then a pair (f, φ) forms a **pattern morphism** from $\mathcal{G}$ to $\mathcal{H}$ if $f : G \to H$ is a map and φ is a residual map from $\mathbb{D}$ to $\mathbb{E}$ satisfying $\varphi \circ \delta = \varepsilon \circ f$, that is, the following diagram is commutative:



In the sequel, we show how our previous considerations apply to pattern structures and breath life into the definition above.

4.3.1 Applications

(1) Let $\mathcal{G}$ be a pattern structure and $\mathcal{H}$ be a pattern setup. If (f, φ) is a pattern morphism from $\mathcal{G}$ to $\mathcal{H}$ with f being surjective, then $\mathcal{H}$ is also a pattern structure.

(2) Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ and $\mathcal{H} = (H, \mathbb{E}, \varepsilon)$ be pattern structures. Also, let (f, φ) be a pattern morphism from $\mathcal{G}$ to $\mathcal{H}$.
To apply the previous theorem, we give the following construction:
f gives rise to an adjunction (α, α^+) between the power set lattices $\mathbf{2}^G := (2^G, \subseteq)$ and $\mathbf{2}^H := (2^H, \subseteq)$ via

$$\alpha : 2^G \to 2^H, X \mapsto fX$$

and

$$\alpha^+ : 2^H \to 2^G, Y \mapsto f^{-1}Y.$$

Further let φ^- denote the residuated map of φ w.r.t. $(\mathbb{E}, \mathbb{D})$, that is, $(\mathbb{E}, \mathbb{D}, \varphi^-, \varphi)$ is a poset adjunction. Then, obviously, $(\mathbb{D}^{\mathrm{op}}, \mathbb{E}^{\mathrm{op}}, \varphi, \varphi^-)$ is a poset adjunction too.

For pattern structures, the following operators are essential:

$$\begin{aligned}
&{}^{\diamond} : 2^G \to D, X \mapsto \inf_{\mathbb{D}} \delta X \\
&{}^{\blacklozenge} : D \to 2^G, d \mapsto \{g \in G \mid d \sqsubseteq \delta g\} \\
&{}^{\square} : 2^H \to E, Z \mapsto \inf_{\mathbb{E}} \varepsilon Z \\
&{}^{\blacksquare} : E \to 2^H, e \mapsto \{h \in H \mid e \sqsubseteq \varepsilon h\}.
\end{aligned}$$

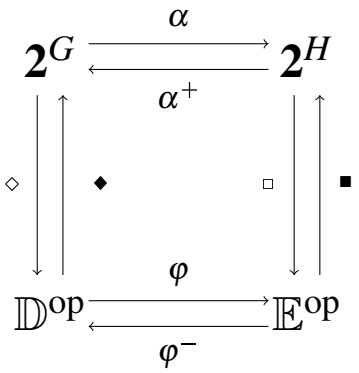
It now follows that (α, φ) forms a morphism from the poset adjunction

$$\mathcal{P} = (\mathbf{2}^G, \mathbb{D}^{\mathrm{op}}, {}^{\diamond}, {}^{\blacklozenge})$$

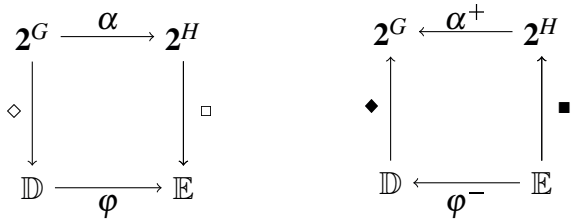
to the poset adjunction

$$\mathcal{Q} = (\mathbf{2}^H, \mathbb{E}^{\mathrm{op}}, {}^{\square}, {}^{\blacksquare}).$$

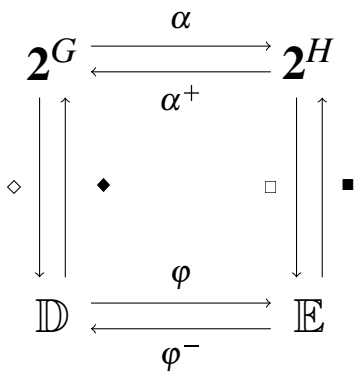
In particular, $(fX)^{\square} = \varphi(X^{\diamond})$ holds for all $X \subseteq G$.
The constructed adjunctions can be illustrated in the following way:



Replacing $\mathbb{D}^{\mathrm{op}}$ by $\mathbb{D}$ and $\mathbb{E}^{\mathrm{op}}$ by $\mathbb{E}$ we receive the following commutative diagrams:



In combination we receive the following diagram of Galois connections and adjunctions between them:



As our next step we recall that the **concept lattice** of $\mathcal{G}$ is given by $\mathbb{B}\mathcal{G} := \mathbb{B}\mathcal{P}$ - similarly, $\mathbb{B}\mathcal{H} := \mathbb{B}\mathcal{Q}$.
We are now able to give an application of Theorem 4.1 to concept lattices of pattern structures: $(\mathbb{B}\mathcal{G}, \mathbb{B}\mathcal{H}, \Phi_{(\alpha,\beta)}, \Phi^+_{(\alpha,\beta)})$ is an adjunction for

$$\Phi_{(\alpha,\beta)} : \mathrm{B}\mathcal{G} \to \mathrm{B}\mathcal{H}, (X,d) \mapsto ((\varphi d)^{\blacksquare}, \varphi d)$$

and

$$\Phi^+_{(\alpha,\beta)} : \mathrm{B}\mathcal{H} \to \mathrm{B}\mathcal{G}, (Z,e) \mapsto (f^{-1}Z, (f^{-1}Z)^{\diamond}).$$

In case f is surjective, $\Phi_{(\alpha,\beta)}$ is surjective too.

Remark: This application implies a generalisation of Proposition 1 in [BKN15b], that is, if Z is an extent in $\mathcal{H}$, then $f^{-1}Z$ is an extent in $\mathcal{G}$, and if d is an intent in $\mathcal{G}$ then φd is an intent in $\mathcal{H}$.

(3) Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure and let κ be a kernel operator on $\mathbb{D}$. Then $\varphi : D \to \kappa D, d \mapsto \kappa d$ forms a residual map from $\mathbb{D}$ to $\kappa\mathbb{D} := \mathbb{D} \mid \kappa D$, and (id_G, φ) is a pattern morphism from $\mathcal{G}$ to $\mathcal{H} := (G, \kappa\mathbb{D}, \varphi \circ \delta)$.

Remark: In [BKN15b], φ is called an o-projection. Application (3) clarifies the role of o-projections for pattern structures.

(4) Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure, and let κ be a residual kernel operator on $\mathbb{D}$. Then, (id_G, κ) is a pattern morphism from $\mathcal{G}$ to $\mathcal{H} := (G, \mathbb{D}, \kappa \circ \delta)$.

Remark: In Chapter 3, κ is also referred to as a residual projection. Application (4) clarifies the role of residual projections for pattern structures.

(5) Generalising [BKN15b] and Chapter 3, we observe that, if $\mathcal{G} = (G, \mathbb{D}, \delta)$ is a pattern structure and φ is a residual map from $\mathbb{D}$ to $\mathbb{E}$, then (id_G, φ) is a pattern morphism from $\mathcal{G}$ to $\mathcal{H} = (G, \mathbb{E}, \varphi \circ \delta)$, satisfying that

$$\Phi : \mathrm{B}\mathcal{G} \to \mathrm{B}\mathcal{H}, (X,d) \mapsto ((\varphi d)^{\blacksquare}, \varphi d)$$

is a surjective residuated map from $\mathbb{B}\mathcal{G}$ to $\mathbb{B}\mathcal{H}$.
In particular, $X^{\square} = \varphi(X^{\diamond})$ holds for all $X \subseteq G$.

Remark: This last application gives a better understanding to properly generalize the concept of projections as discussed in [GK01] and subsequently in [BKN15b, KS11, KKND11, Kuz09, Kuz13, LS15a].

In this chapter we provided a general framework for pattern structures based on the investigation of adjunctions between posets and their morphisms. This yields a general framework for pattern structures. Our investigation included the impact of pattern morphisms on the induced concept lattices. Every representation context of a pattern structure has a concept lattice that is induced by a certain pattern morphism. Morphisms between adjunctions turn out to be of crucial interest for a better understanding of morphisms between concept lattices of pattern structures.

5

Morphisms between Pattern Structures and their Impact on Concept Lattices

This chapter extends the concept of representation contexts and interprets them via morphisms, closely related to o-projections, as introduced and investigated in [BKN15b]. In Chapter 3 and 4, we disussed the meaning of projections of pattern structures, realizing the importance of residual projections. As a matter of fact, our generalization of representation contexts of pattern structures gives rise to residual projections.
Furthermore, we etablish a new view on pattern structures and their representation contexts, which yields to a clearer picture of the work in [GK01].
Parts of this paper were presented at the Workshop "What can FCA do for Artificial Intelligence?" (FCA4AI, 2016) [LS16].

5.1 Extension to Adjunctions and their Concept Posets

The following theorem extends Theorem 4.1 of the last chapter.

Theorem 5.1. Let $(\mathbb{B}\mathcal{P}, \mathbb{B}\mathcal{Q}, \Phi_{(\alpha,\beta)}, \Phi^{+}_{(\alpha,\beta)})$ be the concept poset adjunction induced by (α,β). So, (α,β) is a morphism from a poset adjunction $\mathcal{P} = (\mathbb{P}, \mathbb{S}, \sigma, \sigma^{+})$ to a poset adjunction $\mathcal{Q} = (\mathbb{Q}, \mathbb{T}, \tau, \tau^{+})$ and

$$(\mathbb{B}\mathcal{P}, \mathbb{B}\mathcal{Q}, \Phi_{(\alpha,\beta)}, \Phi^{+}_{(\alpha,\beta)})$$

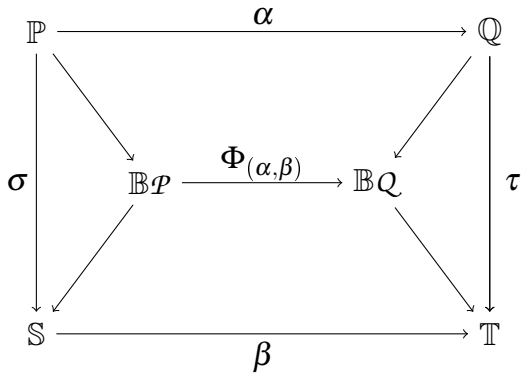
is a poset adjunction for

$$\Phi_{(\alpha,\beta)} : \mathrm{B}\mathcal{P} \to \mathrm{B}\mathcal{Q}, (p,s) \mapsto (\tau^{+}\beta s, \beta s)$$

and

$$\Phi^{+}_{(\alpha,\beta)} : \mathrm{B}\mathcal{Q} \to \mathrm{B}\mathcal{P}, (q,t) \mapsto (\alpha^{+}q, \sigma\alpha^{+}q).$$

The following figure clarifies the facts.



Under these conditions, the following holds:

(1) If α is surjective, then $\Phi_{(\alpha,\beta)}$ is surjective too.

(2) If β is injective, then $\Phi_{(\alpha,\beta)}$ is injective too.

(3) If α is surjective and β is injective, then $\Phi_{(\alpha,\beta)}$ is an isomorphism from $\mathrm{B}\mathcal{P}$ to $\mathrm{B}\mathcal{Q}$.

Proof.

(1) Assume that α is surjective, that is, $\alpha \circ \alpha^+ = id_Q$. Then, for all $(q,t) \in \mathrm{B}\mathcal{Q}$, the second component of

$$(\Phi_{(\alpha,\beta)} \circ \Phi^+_{(\alpha,\beta)})(q,t) \text{ is}$$
$$\beta\sigma\alpha^+ q = \tau\alpha\alpha^+ q = \tau q = t.$$

This yields

$$\Phi_{(\alpha,\beta)} \circ \Phi^+_{(\alpha,\beta)} = id_{\mathrm{B}\mathcal{Q}},$$

that is, $\Phi_{(\alpha,\beta)}$ is surjective.

(2) The first component of

$$(\Phi^+_{(\alpha,\beta)} \circ \Phi_{(\alpha,\beta)})(p,s) \text{ is}$$
$$\alpha^+\tau^+\beta s = \sigma^+\beta^+\beta s = \sigma^+ s = p.$$

Therefore,

$$\Phi^+_{(\alpha,\beta)} \circ \Phi_{(\alpha,\beta)} = id_{\mathrm{B}\mathcal{P}},$$

which yields $\Phi_{(\alpha,\beta)}$ being injective.

(3) If α is surjective and β is injective, then $\Phi_{(\alpha,\beta)}$ and $\Phi^+_{(\alpha,\beta)}$ are naturally inverse by (1) and (2), that is, $\Phi_{(\alpha,\beta)}$ is an isomorphism from $\mathrm{B}\mathcal{P}$ to $\mathrm{B}\mathcal{Q}$.

□

The theorem shows that the morphism allows us to draw conclusions concerning the map between the underlying concept posets. In the following, we are going to examine these ideas more closely.

5.2 The Impact of Pattern Morphism on Concept Lattices

Next, we extend the results of the previous theorem. We are going to use a construction that is already known from Subsection 4.3.1 (Application (2)) of the last chapter.

Theorem 5.2. Let (f,φ) be a pattern morphism from a pattern structure $\mathcal{G} = (G,\mathbb{D},\delta)$ to a pattern structure $\mathcal{H} = (H,\mathbb{E},\varepsilon)$.
To apply the previous theorem, we give the following construction:
f gives rise to an adjunction (α,α^+) between the power set lattice $\mathbf{2}^G := (2^G,\subseteq)$ and the power set lattice $\mathbf{2}^H := (2^H,\subseteq)$ via

$$\alpha : 2^G \to 2^H, X \mapsto fX$$

and

$$\alpha^+ : 2^H \to 2^G, Y \mapsto f^{-1}Y.$$

Further, let φ^- denote the residuated map of φ w.r.t. $(\mathbb{E},\mathbb{D})$, that is,

$$(\mathbb{E},\mathbb{D},\varphi^-,\varphi)$$

is a poset adjunction. Then, obviously,

$$(\mathbb{D}^{\mathrm{op}},\mathbb{E}^{\mathrm{op}},\varphi,\varphi^-)$$

is a poset adjunction too.

As we know for pattern structures the following operators are essential:

$$\begin{aligned}
&{}^{\diamond} : 2^G \to D, X \mapsto \inf_{\mathbb{D}} \delta X\\
&{}^{\blacklozenge} : D \to 2^G, d \mapsto \{g \in G \mid d \sqsubseteq \delta g\}\\
&{}^{\square} : 2^H \to E, Z \mapsto \inf_{\mathbb{E}} \varepsilon Z\\
&{}^{\blacksquare} : E \to 2^H, e \mapsto \{h \in H \mid e \sqsubseteq \varepsilon h\}.
\end{aligned}$$

It now follows that (α,φ) forms a morphism from the poset adjunction

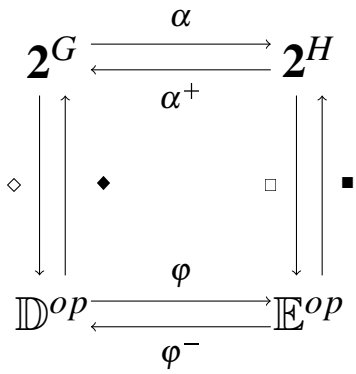
$$\mathcal{P} = (\mathbf{2}^G,\mathbb{D}^{\mathrm{op}},{}^{\diamond},{}^{\blacklozenge})$$

to the poset adjunction

$$\mathcal{Q} = (\mathbf{2}^H,\mathbb{E}^{\mathrm{op}},{}^{\square},{}^{\blacksquare}).$$

In particular, $(fX)^{\square} = \varphi(X^{\diamond})$ holds for all $X \subseteq G$.

We receive the following diagram of adjunctions:

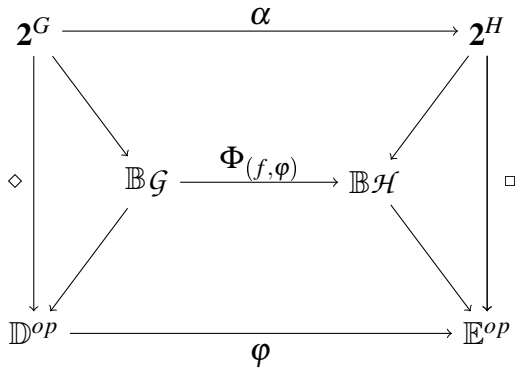


For the following step, we recall that the concept lattice of $\mathcal{G}$ is given by $\mathbb{B}\mathcal{G} := \mathbb{B}\mathcal{P}$ and the concept lattice of $\mathcal{H}$ is $\mathbb{B}\mathcal{H} := \mathbb{B}Q$.
Then Theorem 4.1 yields that the quadruple $(\mathbb{B}\mathcal{G}, \mathbb{B}\mathcal{H}, \Phi_{(f,\varphi)}, \Phi^+_{(f,\varphi)})$ is an adjunction for

$$\Phi_{(f,\varphi)} : \mathrm{B}\mathcal{G} \to \mathrm{B}\mathcal{H}, (X,d) \mapsto ((\varphi d)^{\blacksquare}, \varphi d)$$

and

$$\Phi^+_{(f,\varphi)} : \mathrm{B}\mathcal{H} \to \mathrm{B}\mathcal{G}, (Z,e) \mapsto (f^{-1}Z, (f^{-1}Z)^{\diamond}).$$



As per Theorem 5.1 the following holds:

(1) If f is surjective, then $\Phi_{(f,\varphi)}$ is surjective too.

(2) If φ is injective, then $\Phi_{(f,\varphi)}$ is injective too.

(3) If f is surjective and φ is injective, then $\Phi_{(f,\varphi)}$ is an isomorphism from $\mathbb{B}\mathcal{G}$ to $\mathbb{B}\mathcal{H}$.

Remark: To apply Theorem 4.1 in the above theorem, we point out that $\Phi_{(f,\varphi)} = \Phi_{(\alpha,\varphi)}$ holds.

The following definition and theorem are results of our investigations.

Definition 5.1 (induced pattern structure)**.** Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure and let

$$\mathcal{G}^\bullet = (G, \mathbb{C}_\mathcal{G}, \delta^\bullet)$$

be the pattern structure induced by $\mathcal{G}$ via

$$\delta^\bullet : G \to C_\mathcal{G},\ g \mapsto \delta g \text{ with}$$

$$\mathrm{C}_\mathcal{G} := \{\inf_\mathbb{D} \delta X \mid X \subseteq G\} \text{ and } \mathbb{C}_\mathcal{G} := \mathrm{C}_\mathcal{G}|\mathbb{D}.$$

Then, we call $\mathcal{G}^\bullet = (G, \mathbb{C}_\mathcal{G}, \delta^\bullet)$ the by $\mathcal{G}$ **induced pattern structure**.

Theorem 5.3. Let $\mathcal{G} = (G, \mathbb{D}, \delta)$, $\mathcal{H} = (H, \mathbb{E}, \varepsilon)$ be pattern structures and let $\mathcal{G}^\bullet = (G, \mathbb{C}_\mathcal{G}, \delta^\bullet)$ be the by $\mathcal{G}$ induced pattern structure. Then, it follows that $\mathbb{B}\mathcal{G}^\bullet = \mathbb{B}\mathcal{G}$. Further, let (f, φ) be a pattern morphism from $\mathcal{G}^\bullet$ to $\mathcal{H}$. Then, with the notation introduced in the previous theorem, the map $\Phi_{(f,\varphi)}$ from $\mathbb{B}\mathcal{G}$ to $\mathbb{B}\mathcal{H}$ is residuated. If f is surjective, then so is $\Phi_{(f,\varphi)}$; if φ is injective, then so is $\Phi_{(f,\varphi)}$. If f is surjective and φ is injective, then $\Phi_{(f,\varphi)}$ is an isomorphism from $\mathbb{B}\mathcal{G}$ to $\mathbb{B}\mathcal{H}$.

In the following we are going to investigate the connection between formal contexts and pattern structures. We will need the definition and the theorem below.

Definition 5.2 (representation context)**.** The **representation context** of a pattern structure $\mathcal{G} = (G, \mathbb{D}, \delta)$ w.r.t. a subset M of D is given by

$$\mathbb{K}(\mathcal{G}, M) := (G, M, I) \text{ with}$$
$$I := \{(g, m) \in G \times M \mid m \sqsubseteq \delta g\}.$$

Theorem 5.4. Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure and let M be a subset of D.
The associated pattern structure of the representation context $\mathbb{K}(\mathcal{G}, M)$ is given by

$$\mathcal{H} := (G, \mathbf{2}^M, \varepsilon),$$

with

$$\varepsilon : G \to 2^M, g \mapsto^{\downarrow}_M \delta g,$$

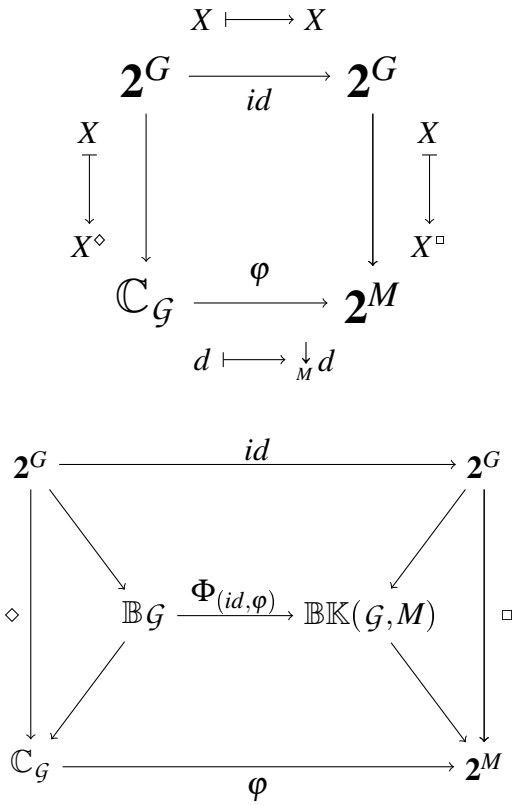
where

$$^{\downarrow}_M d := \{m \in M \mid m \sqsubseteq d\} \text{ for all } d \in D.$$

In particular, the concept lattice of $\mathbb{K}(\mathcal{G}, M)$ is given by $\mathbb{B}\mathbb{K}(\mathcal{G}, M) = \mathbb{B}\mathcal{H}$.
Using the notation from the previous theorem, (id, φ) is a pattern morphism from $\mathcal{G}^\bullet$ to $\mathcal{H}$ for

$$\varphi : C_\mathcal{G} \to 2^M, x \mapsto^{\downarrow}_M x.$$

Furthermore, the map $\Phi_{(id,\varphi)}$ from $\mathbb{B}\mathcal{G}$ to $\mathbb{B}\mathcal{H} = \mathbb{BK}(\mathcal{G},M)$ is a residuated surjection. In case M is join-dense w.r.t. $\mathbb{C}_{\mathcal{G}}$ (that is, φ is injective), $\Phi_{(id,\varphi)}$ is an isomorphism from $\mathbb{B}\mathcal{G}$ to $\mathbb{BK}(\mathcal{G},M)$.



Remark: Based on the paradigm of concept morphisms, the previous theorem extends and sheds new light on Theorem 1 of [GK01]. We generalize the definition of representation contexts, introduced in [GK01], by allowing an arbitrary subset M of patterns of the underlying pattern structure $\mathcal{G}$ as an attribute set of the representation context $\mathbb{K}(\mathcal{G},M)$. It then turns out that $\mathbb{K}(\mathcal{G},M)$ has a concept lattice which is induced by a morphism on $\mathcal{G}^\bullet$. More explicitly, there is a morphism on $\mathcal{G}^\bullet$ to the pattern structure of $\mathbb{K}(\mathcal{G},M)$ which induces a residuated surjection from the concept lattice of $\mathcal{G}$ to the concept lattice of $\mathbb{K}(\mathcal{G},M)$. In case M is join-dense with respect to $\mathcal{G}$, the morphism between the concept lattices is an isomorphism (see also Theorem 1 of [GK01]). Our extension of the concept of representation context gives rise to various constructions of o-projections (as introduced in [BKN15b]) on $\mathcal{G}^\bullet$.

In this chapter we shed new light on the concepts presented in [GK01, KS11]. Beyond this, we showed that poset adjunctions and their morphisms provide an elegant theoretical tool for discussing possibilities of complexity reduction of pattern structures and their morphisms.
As a major insight, morphisms between adjunctions are apparently highly relevant for the analysis of morphisms between concept lattices of pattern structures.

6

Viewing Morphisms between Pattern Structures via their Concept Lattices and via their Representations

Our approach in this chapter is a continuation of Chapter 5 and discusses the theoretical framework for pattern structures and their morphisms with respect to their concept lattices and their representations. In particular, we investigate the possible complexity reductions beyond projections and o-projections, as studied in the previous chapters and [BKN15b].
As a novel idea, we present the theoretical conditions of complexity reduction for adjunctions and, subsequently, for pattern morphisms and their representations. More preciseley, our Theorem 6.4 clarifies the theoretical background of Theorem 2 in [GK01] and its adjustments (e.g. see [KS11] Theorem 3).
The results of this chapter were presented on the International Symposium on Methologies for Intelligent Systems (ISMIS) 2017 [LS17].

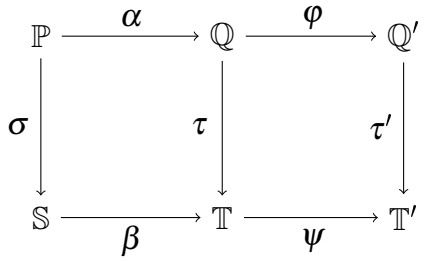
6.1 Concatenation of Morphisms

In the following theorems we investigate the theoretical background of a reduction of complexity for pattern structures and, even more, of pattern morphisms. Our results follow a top-down strategy starting with a general setup of poset adjunctions (see Theorem 6.1); then we specify the situation for pattern structures (see Theorem 6.2) and their representations. In particular, we discuss how morphisms between pattern structures induce morphisms between their representations (see Theorem 6.3 and Theorem 6.4). Our Theorem 6.4 clarifies the theoretical background of Theorem 2 in [GK01] and its adjustments (e.g. see [KS11] Theorem 3).
As mentioned before, we start with a general setup. So far, we looked at the interaction of two poset adjunctions. Hereinafter, we will gain new insights by extending the number of poset adjunctions.

Definition 6.1 (concatenation of morphisms)**.** Let (α,β) be a morphism from a poset adjunction $\mathcal{P}=(\mathbb{P},\mathbb{S},\sigma,\sigma^+)$ to a poset adjunction $\mathcal{Q}=(\mathbb{Q},\mathbb{T},\tau,\tau^+)$ and (φ,ψ) be a morphism from $\mathcal{Q}$ to a poset adjunction $\mathcal{Q}'=(\mathbb{Q}',\mathbb{T}',\tau',\tau'^+)$ then the **concatenation** of (α,β) with (φ,ψ) is given by

$$(\varphi,\psi)\circ(\alpha,\beta) := (\varphi\circ\alpha,\psi\circ\beta).$$

The figure clarifies the facts.

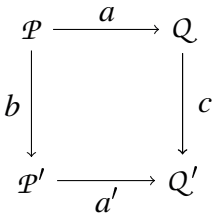


Remark: The concatenation of morphisms between poset adjunctions is again a morphism between poset adjunctions.

Definition 6.2 (commutative square of morphisms)**.** If $\mathcal{P}=(\mathbb{P},\mathbb{S},\sigma,\sigma^+)$, $\mathcal{Q}=(\mathbb{Q},\mathbb{T},\tau,\tau^+)$, $\mathcal{P}'=(\mathbb{P}',\mathbb{S}',\sigma',\sigma'^+)$ and $\mathcal{Q}'=(\mathbb{Q}',\mathbb{T}',\tau',\tau'^+)$ are poset adjunctions, and

$$\begin{aligned}&\mathcal{P}\xrightarrow{a}\mathcal{Q},\\ &\mathcal{P}'\xrightarrow{a'}\mathcal{Q}',\\ &\mathcal{P}\xrightarrow{b}\mathcal{P}' \text{ and}\\ &\mathcal{Q}\xrightarrow{c}\mathcal{Q}'\end{aligned}$$

are morphisms such that $c\circ a = a'\circ b$ holds, we will say that (a,c,b,a') is a **commutative square** of morphisms between poset adjunctions.

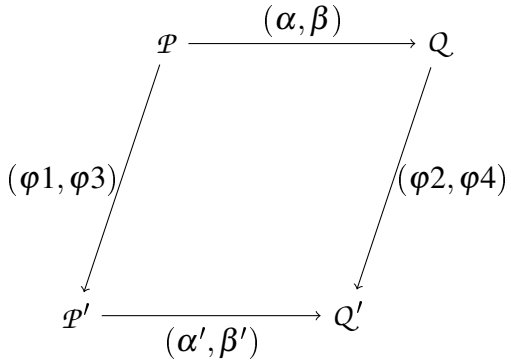


We are now able to formulate a first statement on commutative squares of morphisms between poset adjunctions.

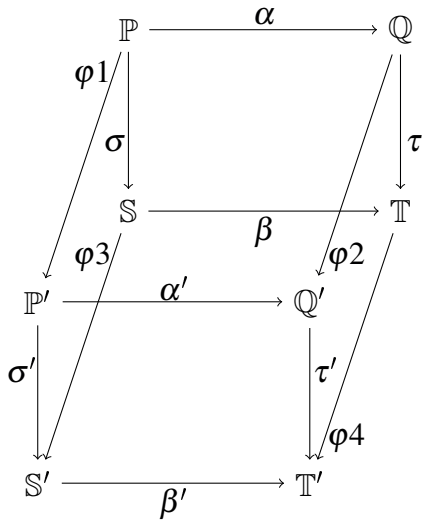
Theorem 6.1. A commutative square of morphisms between poset adjunctions induces a commutative square of residuated maps between their concept posets.
More explicitly, let

$$\begin{aligned}
&\mathcal{P} = (\mathbb{P}, \mathbb{S}, \sigma, \sigma^+),\\
&\mathcal{Q} = (\mathbb{Q}, \mathbb{T}, \tau, \tau^+),\\
&\mathcal{P}' = (\mathbb{P}', \mathbb{S}', \sigma', \sigma'^+) \text{ and}\\
&\mathcal{Q}' = (\mathbb{Q}', \mathbb{T}', \tau', \tau'^+)
\end{aligned}$$

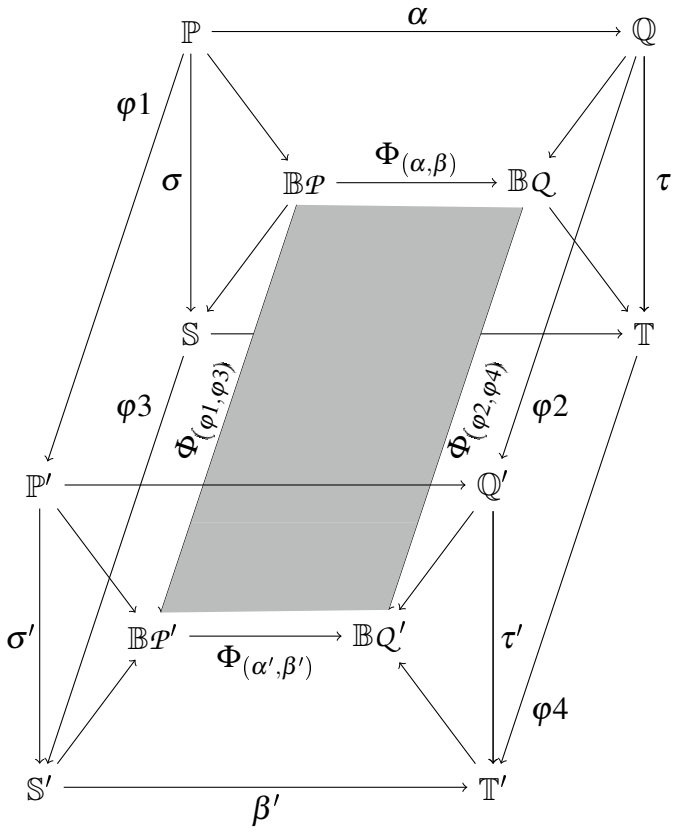
be poset adjunctions and let



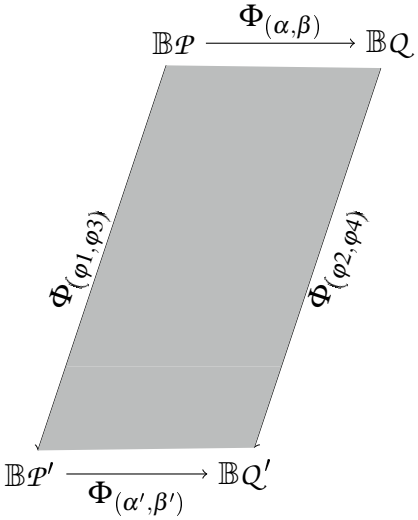
be a commutative square of morphisms between poset adjunctions, that is, the six sides of the following cube consist of commutative squares of residuated maps.



Then, the following diagram of induced residuated maps between the concept posets of the involved poset adjunctions (as constructed in Theorem 4.1) is commutative:



In particular, the following residuated maps form a commutative square between concept posets.



Proof. Our claim is $\Phi_{(\varphi 2,\varphi 4)} \circ \Phi_{(\alpha,\beta)} = \Phi_{(\alpha',\beta')} \circ \Phi_{(\varphi 1,\varphi 3)}$. So, let $(p,s) \in \mathbb{B}\mathcal{P}$, resulting in

$$\Phi_{(\varphi 2,\varphi 4)}(\Phi_{(\alpha,\beta)}(p,s)) = \Phi_{(\varphi 2,\varphi 4)}(\tau^{+}\beta s, \beta s) = (\tau'^{+}\varphi 4\beta s, \varphi 4\beta s)$$

and

$$\Phi_{(\alpha',\beta')}(\Phi_{(\varphi 1,\varphi 3)}(p,s)) = \Phi_{(\alpha',\beta')}(\sigma^{+}\varphi 3s, \varphi 3s) = (\tau'^{+}\beta'\varphi 3s, \beta'\varphi 3s).$$

Since $\varphi 4 \circ \beta = \beta' \circ \varphi 3$, we conclude

$$\begin{aligned}(\Phi_{(\varphi 2,\varphi 4)} \circ \Phi_{(\alpha,\beta)})(p,s) &= (\tau'^{+}\varphi 4\beta s, \varphi 4\beta s)\\ &= (\tau'^{+}\beta'\varphi 3s, \beta'\varphi 3s)\\ &= (\Phi_{(\alpha',\beta')} \circ \Phi_{(\varphi 1,\varphi 3)})(p,s).\end{aligned}$$

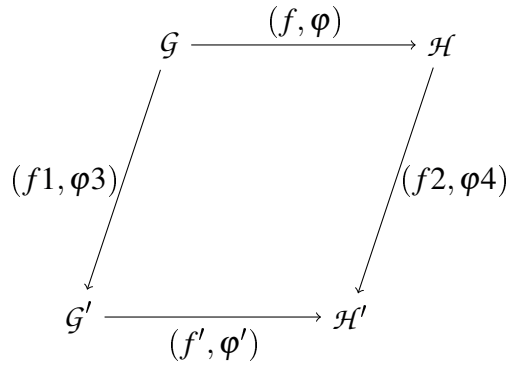
□

From the general setup with poset adjunctions, we now come to a more specific case using pattern structures.

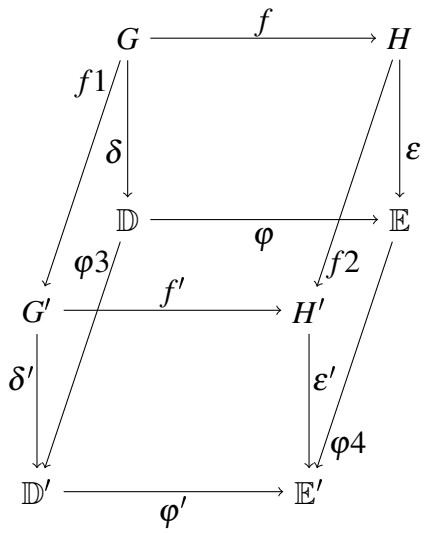
Theorem 6.2. A commutative square of pattern morphisms between pattern structures induces a commutative square of residual maps between their concept lattices.
More explicitly, let

$$\begin{aligned}&\mathcal{G} = (G, \mathbb{D}, \delta),\\ &\mathcal{H} = (H, \mathbb{E}, \varepsilon),\\ &\mathcal{G}' = (G', \mathbb{D}', \delta') \text{ and}\\ &\mathcal{H}' = (H', \mathbb{E}', \varepsilon')\end{aligned}$$

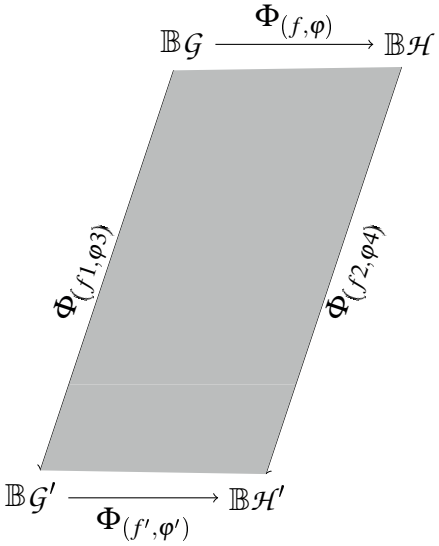
be pattern structures, and let



be a commutative square of pattern morphisms, that is, the six sides of the following cube are commutative squares.

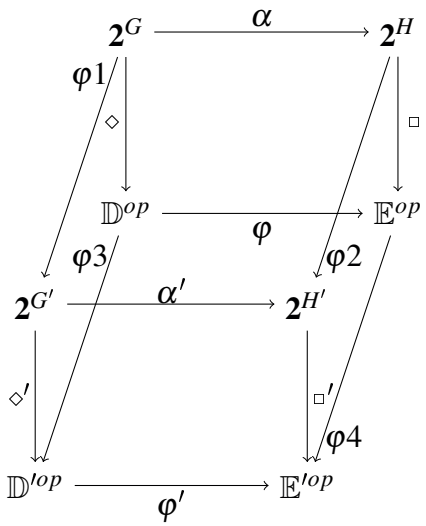


Then,



forms a commutative square of residuated maps between concept lattices of pattern structures.

Proof. The six sides of the following cube are commutative squares of residuated maps.



We now apply Theorem 6.1 and receive

$$\Phi_{(\varphi 2,\varphi 4)} \circ \Phi_{(\alpha,\varphi)} = \Phi_{(\alpha',\varphi')} \circ \Phi_{(\varphi 1,\varphi 3)},$$

that is,

$$\Phi_{(f 2,\varphi 4)} \circ \Phi_{(f,\varphi)} = \Phi_{(f',\varphi')} \circ \Phi_{(f 1,\varphi 3)}.$$

□

The following definition provides a reduction of complexity on the set of patterns for a given pattern structure.

Definition 6.3 (M-representation)**.** If $\mathcal{G} = (G, \mathbb{D}, \delta)$ is a pattern structure and $M \subseteq D$, then the M-**representation** of $(\mathcal{G}, M)$ is defined as

$$\mathcal{P}(\mathcal{G}, M) := (G, \mathbf{2}^M, \underline{\delta}) \text{ with}$$
$$\underline{\delta} : G \to 2^M, g \mapsto \downarrow_M \delta g := \{m \in M \mid m \sqsubseteq \delta g\},$$

that is, $\underline{\delta} g$ is the set of all subpattterns of $g \in G$ which are contained in M.

Remark: We want to point out that, for every formal context, its concept lattice coincides with the concept lattice of its associated pattern structure.

(i) For every formal context $\mathbf{K} = (G, M, I)$ is its associated pattern structure (introduced in Definition 2.2) is given by

$$\mathcal{P}\mathbf{K} := (G, \mathbf{2}^M, \delta) \text{ with}$$
$$\delta : G \to 2^M, g \mapsto \{m \in M \mid gIm\}.$$

Then, obviously, the concept lattices of $\mathbf{K}$ and $\mathcal{P}\mathbf{K}$ coincide.

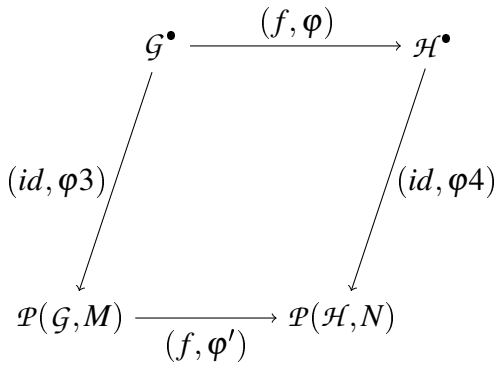
(ii) For a pattern structure $\mathcal{G} = (G, \mathbb{D}, \delta)$ and a subset M of D the representation context of $\mathcal{G}$ over M (as defined in Definition 5.2) is given by

$$\mathbf{K}(\mathcal{G}, M) := (G, M, I) \text{ with}$$
$$I := \{(g, m) \in G \times M \mid m \sqsubseteq \delta g\}.$$

Then the concept lattices of $\mathcal{P}(\mathcal{G}, M)$ and $\mathbf{K}(\mathcal{G}, M)$ coincide and will be denoted by $\mathbb{B}(\mathcal{G}, M)$.

We want to point out that representation contexts as defined in [GK01] are in this chapter discussed within the framework of M-representations of pattern structures. The following two theorems use induced pattern structures, introduced in Definition 5.1, and the knowledge we gained about them in Theorem 5.3. Thereby we demonstrate, which impact a complexity reduction on the set of patterns has on the corresponding concept lattices of the pattern structures. Theorem 6.4 is a special case of Theorem 6.3, where both pattern structures have the same set of patterns. Nevertheless, due to its high relevance for real world applications, we are going to display Theorem 6.4 as well.

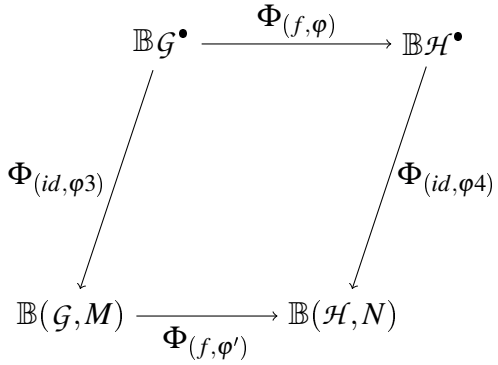
Theorem 6.3. Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure and $\mathcal{G}^\bullet = (G, \mathbb{C}_\mathcal{G}, \delta^\bullet)$ the by $\mathcal{G}$ induced pattern structure. In addition let $\mathcal{H} = (H, \mathbb{E}, \varepsilon)$ be a pattern structure and $\mathcal{H}^\bullet = (H, \mathbb{C}_\mathcal{H}, \varepsilon^\bullet)$ the by $\mathcal{H}$ induced pattern structure. Furthermore, let $M \subseteq D$ and $N \subseteq E$. Also, let



with

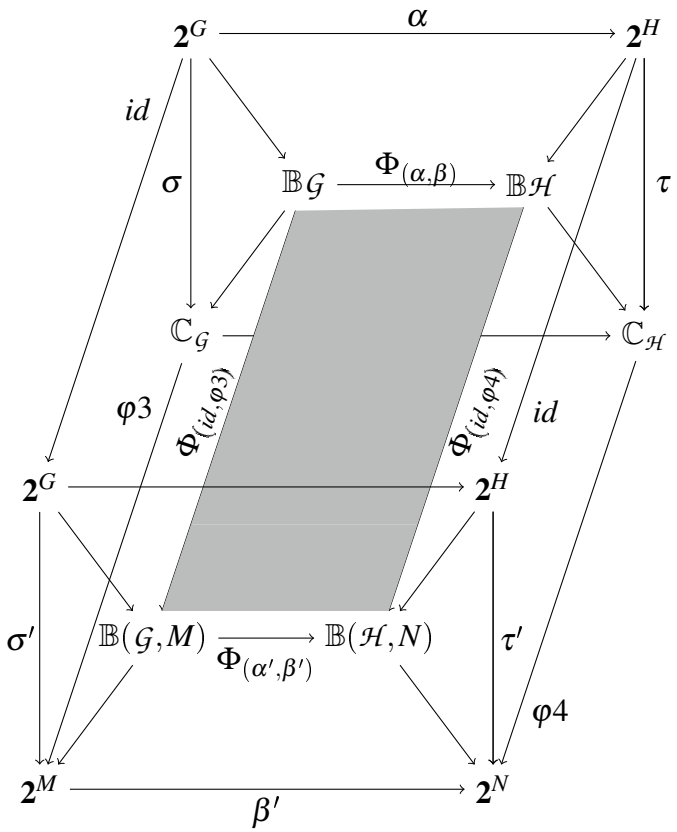
$$\varphi 3 : C_\mathcal{G} \to 2^M, d \mapsto \downarrow_M d$$
$$\varphi 4 : C_\mathcal{H} \to 2^N, e \mapsto \downarrow_N e$$

be a commutative square of pattern morphisms. Then, more explicitly,



is a commutative square of residuated maps between concept lattices of pattern structures.

Proof. The proof follows by Theorem 6.2 by considering the following commutative diagram:



□

As mentioned earlier, the next theorem explains the theoretical background of Theorem 2 in [GK01] and its adjustments (e.g. see [KS11] Theorem 3).

Theorem 6.4. Let $\mathcal{G} = (G, \mathbb{D}, \delta)$ be a pattern structure and $\mathcal{G}^\bullet = (G, \mathbb{C}_\mathcal{G}, \delta^\bullet)$ the by $\mathcal{G}$ induced pattern structure. In addition let $\mathcal{H} = (H, \mathbb{E}, \varepsilon)$ be a pattern structure and $\mathcal{H}^\bullet = (H, \mathbb{C}_\mathcal{H}, \varepsilon^\bullet)$ the by $\mathcal{H}$ induced pattern structure. Furthermore, let M, N be sets with $N \subseteq M \subseteq D$. Also, let

$$\begin{array}{ccc} \mathcal{G}^\bullet & \xrightarrow{(id,\varphi)} & \mathcal{H}^\bullet \\ {\scriptstyle (id,\varphi 3)}\big\downarrow & & \big\downarrow{\scriptstyle (id,\varphi 4)} \\ \mathcal{P}(\mathcal{G}, M) & \xrightarrow[(id,\varphi')]{} & \mathcal{P}(\mathcal{H}, N) \end{array}$$

with

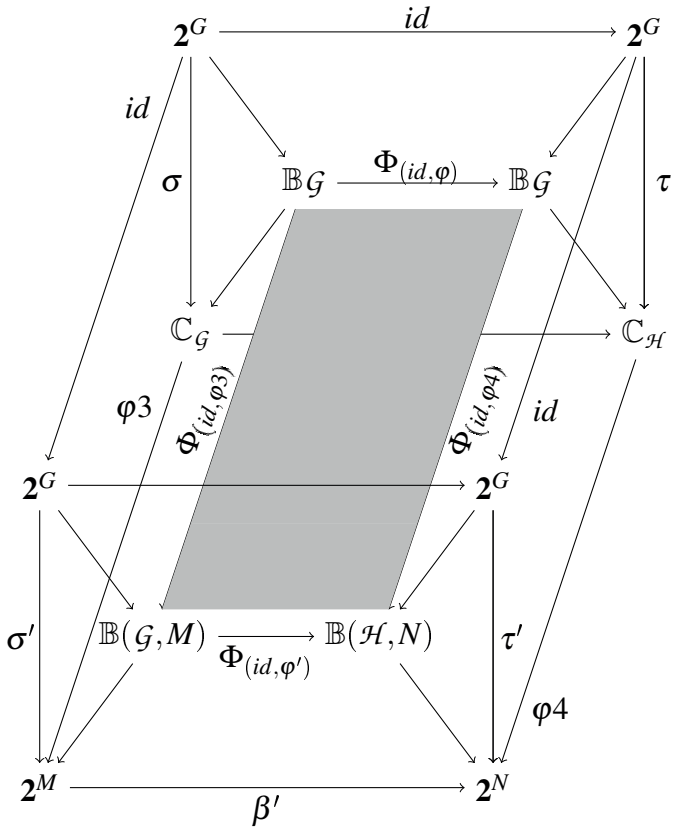
$$\begin{aligned} \varphi 3 &: C_\mathcal{G} \to 2^M, d \mapsto {}^{\downarrow}_{M}\, d \\ \varphi 4 &: C_\mathcal{H} \to 2^N, e \mapsto {}^{\downarrow}_{N}\, e \\ \varphi' &: 2^M \to 2^N, Y \mapsto Y \cap N \end{aligned}$$

be a commutative square of pattern morphisms. Then,

$$\begin{array}{ccc} \mathbb{B}\mathcal{G}^\bullet & \xrightarrow{\Phi_{(f,\varphi)}} & \mathbb{B}\mathcal{H}^\bullet \\ {\scriptstyle \Phi_{(id,\varphi 3)}}\big\downarrow & & \big\downarrow{\scriptstyle \Phi_{(id,\varphi 4)}} \\ \mathbb{B}(\mathcal{G}, M) & \xrightarrow[\Phi_{(f,\varphi')}]{} & \mathbb{B}(\mathcal{H}, N) \end{array}$$

is a commutative square of residuated maps between concept lattices of pattern structures.

Proof. The proof follows by Theorem 6.2 by considering the following commutative diagram:



□

In this chapter we put new light on the work in [GK01]. The new framework of representation contexts and commutative pattern morphisms, presented here, leads to a better and deeper understanding of the theory in [GK01].

In continuation of Chapter 4 and Chapter 5, we investigated pattern structures and their morphisms aiming to provide a theoretical background for complexity reduction. Our results followed a top-down strategy starting with a general setup of poset adjunctions, later specifying the situation for pattern structures and their representations. In particular, we discussed how morphisms between pattern structures induce morphisms between their representations. Morphisms between adjunctions turned out to be of crucial interest for a clearer picture of morphisms between concept lattices of pattern structures.

Part II

Applications

7 A Link to Random Forests

The following chapter shows that decision trees and random forests can be described via pattern structures. This new perspective leads to a better understanding of random forests and delivers a new way of analysing complex data with the help of pattern structures. To apply our findings, we will try to solve a data mining problem, using the data set from [CCA+09] in order to train an algorithm to predict the quality of red wines. However, at first, we will examine the data set more closely.
Parts of this chapter and a similar example have been published in [LS20a].

7.1 The Red Wine Data Set

To apply our results on a public data set, we chose the red wine data set from [CCA+09]. There are 1599 examples of wines, described by 11 numerical attributes.

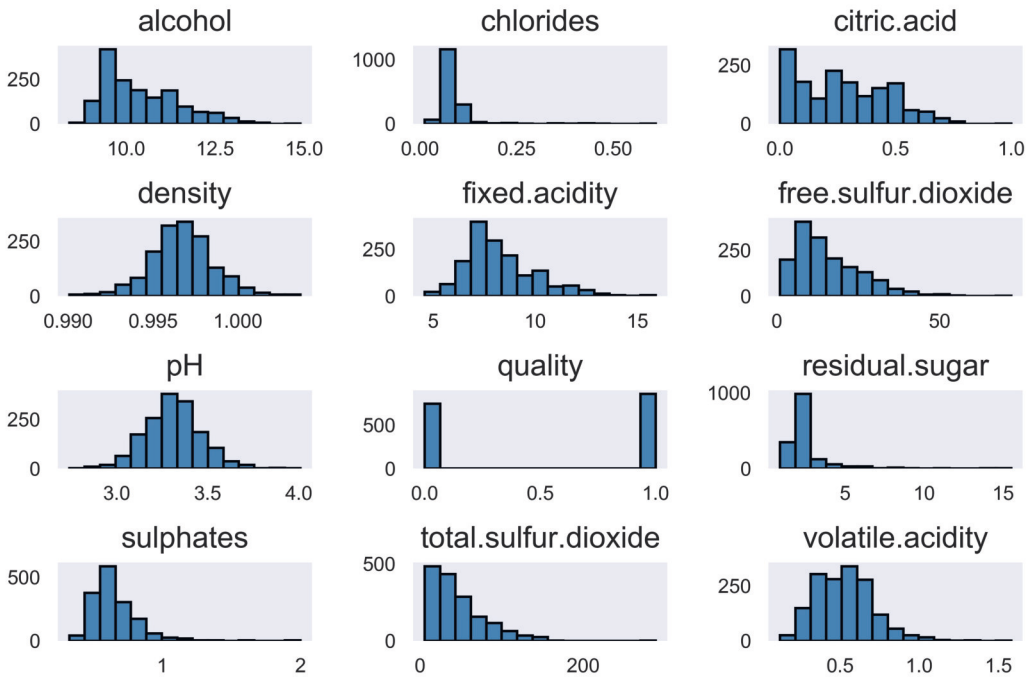


Figure 7.1: All attributes of the red wine data set

The input includes objective tests and the output (quality) is based on sensory data (median of at least 3 evaluations made by wine experts). Each expert graded the wine quality between 0 (very bad) and 10 (very excellent). For our purpose, we established a binary distinction where every wine with a quality score above 5 is classified "good" and all wines below as "bad", leading to a set of 855 positive and 744 negative examples. The following figures show the distribution of good and bad wines for every attribute.

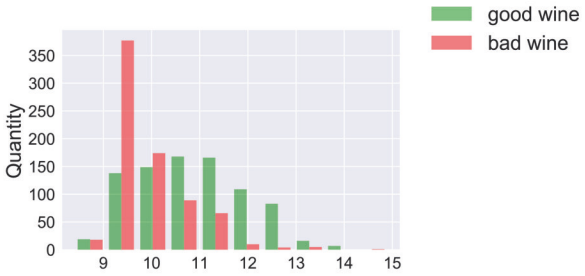


Figure 7.2: alcohol

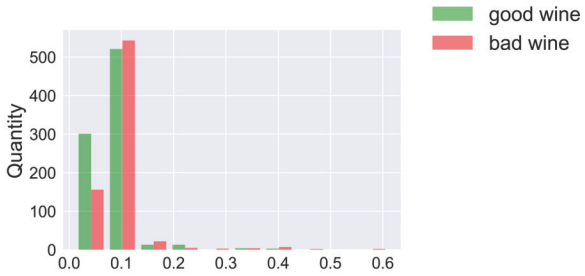


Figure 7.3: chloride

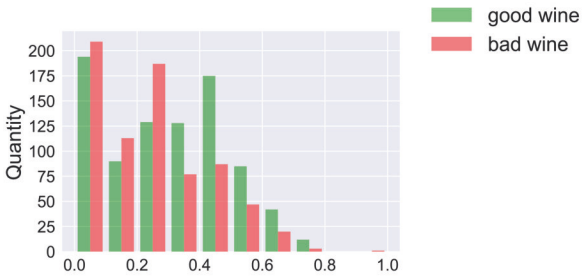


Figure 7.4: acid

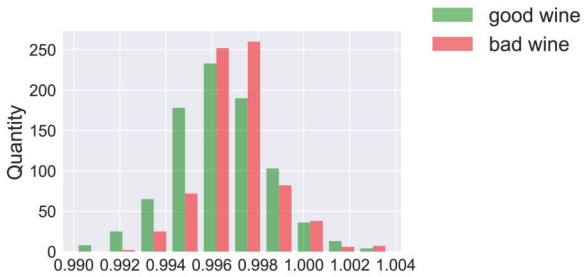


Figure 7.5: density

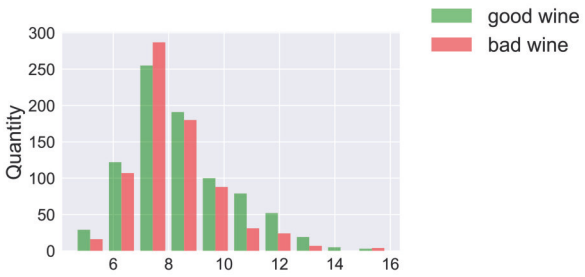


Figure 7.6: acidity

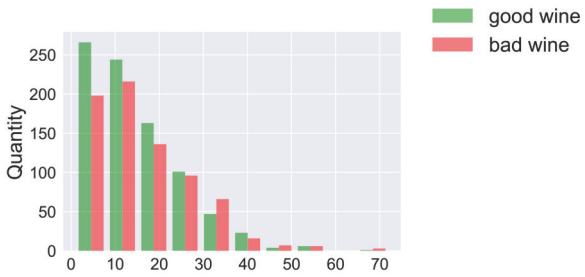


Figure 7.7: sulfur dioxide

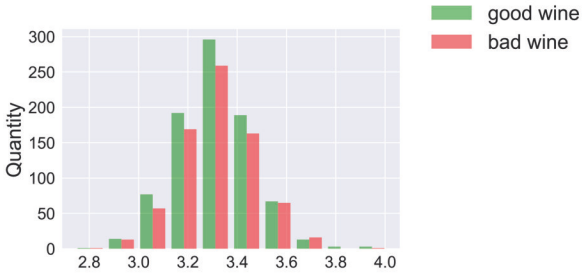


Figure 7.8: ph-value

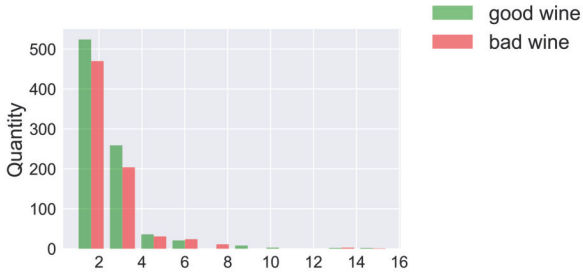


Figure 7.9: residual sugar

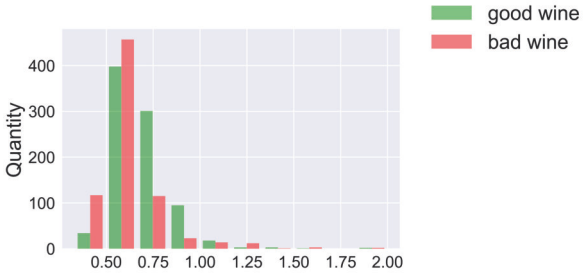


Figure 7.10: sulphates

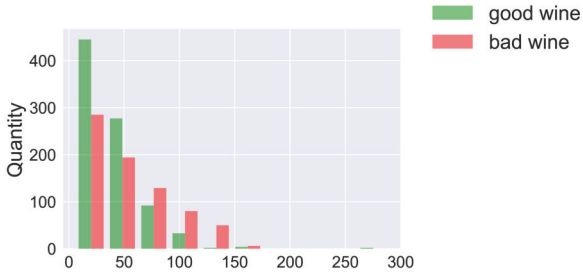


Figure 7.11: sulfur dioxide

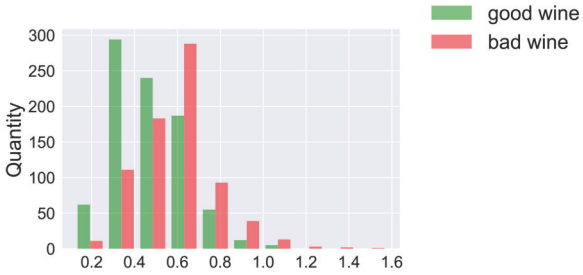


Figure 7.12: volatile acidity

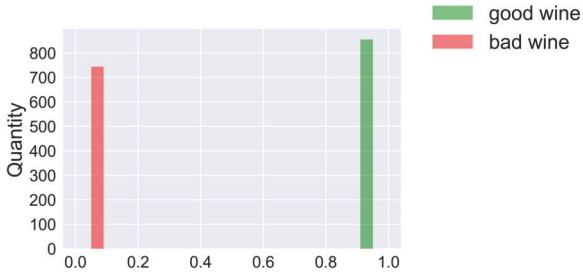


Figure 7.13: quality

7.2 Decision Trees as a Data Mining Tool

In this section, we give a short overview of decision trees and random forests. For an introduction to this subject, we recommend [BFSO84, SL91]. Decision trees are an important data mining tool (for example see [Mah05], [DUA06], [SLT+03], [BD16], [RGGR+12]) used to predict classes of data sets. Many applications of decision trees build a model on a so-called training set and then try to predict classes of unseen data (test sets). As an example, we split our red wine data set of the last chapter into a training set (75% of the data) and a test set (25% of the data). Most of the facts presented in this section are well known, but our order-theory based definitions of decision trees and random forests justify this section here, and not in the preliminaries part. We begin with the construction of a decision tree.

Construction 7.1 (decision tree)**.** We start at the root node and split the data on the feature that results in the best splitting measure (e.g. least gini impurity). In an iterative process, we repeat this splitting procedure at each child node until the nodes are pure (contain only one class) or a termination rule is fulfilled. Nodes that do not split the data any further are called leaves.

The construction mentioned the gini impurity as a splitting measure. The following definition gives a brief insight into the world of splitting criteria.

Definition 7.1 (gini impurity measure, gini splitting criterion)**.** The **gini impurity** at a node t is defined as

$$i(t) = \sum_{i,j} C(i \mid j)p(i \mid t)p(j \mid t),$$

where $C(i \mid j)$ is the cost of mis-classifying a class-j case as a class-i case (clearly, $C(j \mid j) = 0$) and $p(i \mid t)$ is the probability of a case in class i given that it falls into node t. The **gini splitting criterion** is the decrease of impurity, defined as

$$\triangle i(s,t) = i(t) - p_L i(t_L) - p_R i(t_R)$$

where p_L and p_L are probabilities of sending a case to the left child node t_L and to the right child node t_R respectively.

Besides the gini splitting criterion, there are other splitting criterions, such as: entropy, twoing criterion, ordered twoing criterion.... More information and examples on splitting criteria can be found in [RM05].

The following definition delivers a concrete definition of a decision tree.

Definition 7.2 (decision tree)**.** A **finite tree** is defined as a triple $\mathbb{T} := (T, \leqslant_{\mathbb{T}}, 0_{\mathbb{T}})$ consisting of a finite poset $(T, \leqslant)$ with least element $0_{\mathbb{T}}$. This least element will be denoted as **root node**. Every principal downset is totally ordered. The maximal elements of $\mathbb{T}$ will be called **leaves** of $\mathbb{T}$; let $Le\mathbb{T}$ denote the set of leaves of $\mathbb{T}$. A triple $\mathcal{T} := (G, \mathbb{T}, \lambda)$ will be called a **finite decision**

tree or short **decision tree** if G is a finite set, $\mathbb{T} := (T, \leqslant_{\mathbb{T}}, 0_{\mathbb{T}})$ is a finite tree and $\lambda : G \to LeT$ is a surjective map. The map

$$\tau : T \to LeT, t \mapsto \{b \in LeT \mid t \leqslant b\}$$

allocates the set of leaves to a node, where the node is less than or equal to the leaf (in the sense of $\leqslant_{\mathbb{T}}$). We call

$$\mathbb{T}^c := (T^c, \leqslant_{\mathbb{T}}, 0_{\mathbb{T}}) \text{ with } T^c := T \cup \{1_{\mathbb{T}}\}$$

the **tree completion** of $\mathbb{T}$ and $\mathbb{T}^c$ a **complete tree**. This leads to a map

$$\overline{\lambda} : G \to T^c,\ g \mapsto \lambda g.$$

In this context, we will refer to $\mathcal{T}^s := (G, \mathbb{T}, \mathbb{T}^c, \lambda, \overline{\lambda}, \tau)$ as a **decision tree setup**.

Remarks:

(1) A tree completion $\mathbb{T}^c$ is the smallest complete lattice which contains $\mathbb{T}$.

(2) The λ induced partition of G is given by

$$IP_\lambda : LeT \to 2^G,\ b \mapsto \lambda^{-1} b$$

To breathe life into the construction and get some insights on the red wine data set of this section, we use it to build a decision tree. More precisley, we build a decision tree on the training set and use it to predict the classes of the test set. For better readability, we set the depth of the tree to three, which means that the longest path from the root node to a leaf has length three. This leads to the following.

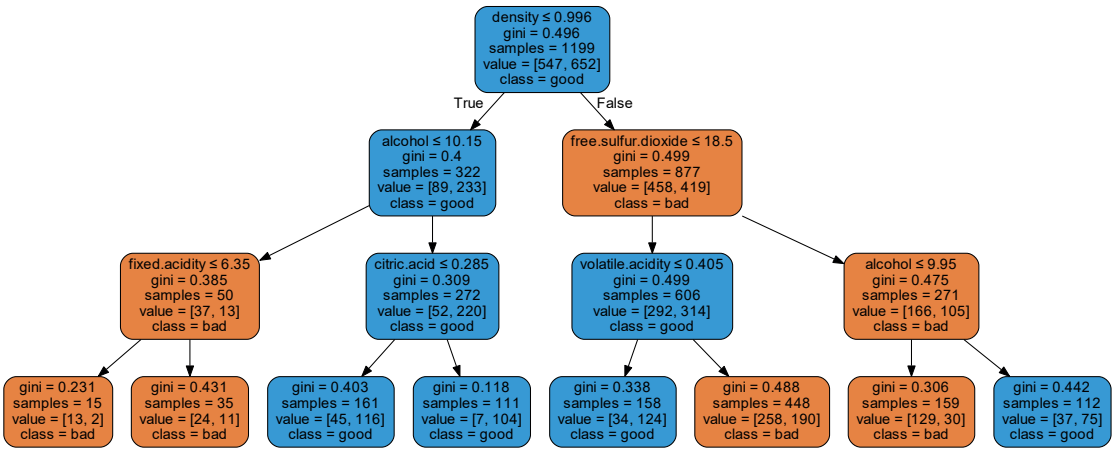


Figure 7.14: Decision tree built on the red wine training set

We use the decision tree to predict the classes of the test set. This delivers the following results.

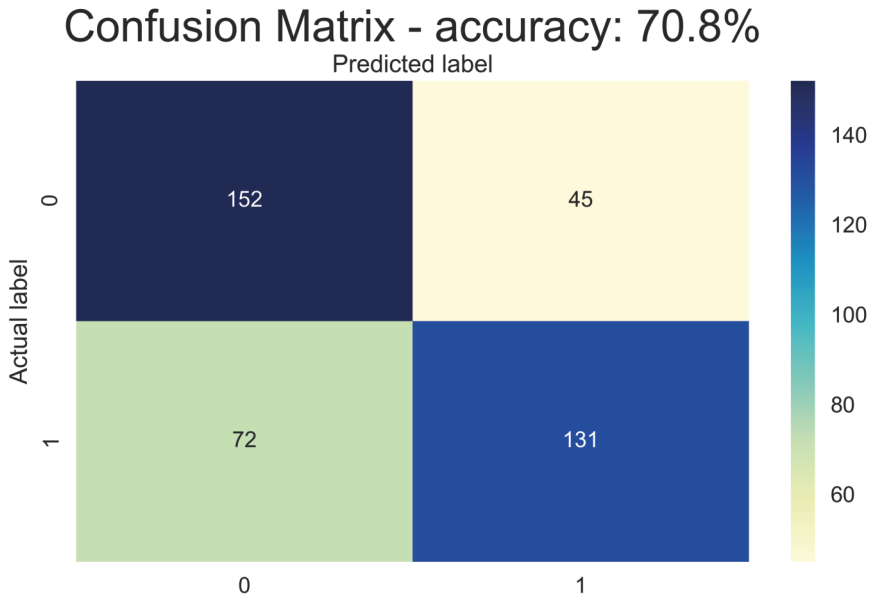


Figure 7.15: Predicting the classes of the test set

As a reading example: the 131 in the down right corner are predicted (x-axis) as good wines because they are predicted as 1, and actually (y-axis) are good wines since the actual label is 1. 131 wines are predicted as good and actually are good. 152 are predicted as bad and actually are bad (top left). So 283 (125+158) of the 400 wines are predicted correctly, which leads to an accuracy of 70,8%. In [Bre01] Leo Breiman introduced random forests, which achieve significant improvements in classification accuracy by growing an ensemble of decision trees and letting them vote for the most popular class. Each tree in the forest is trained on a random subset of data points and features. This is the reason why this model is called 'random' forest. In this thesis we use the following definition of a random forest.

Definition 7.3 (random forest)**.** Let $\mathcal{T}_i^s := (G, \mathbb{T}_i, \mathbb{T}_i^c, \lambda_i, \overline{\lambda_i}, \tau_i)$ be a decision tree setup for all $i \in I$, then we call

$$\mathbb{F} := (G, \mathbb{T}_{prod}, \lambda) \text{ with}$$

$$T_{prod} := \prod_{i \in I} T_i,\ \mathbb{T}_{prod} := \prod_{i \in I} \mathbb{T}_i \text{ and } \lambda : G \to \prod_{i \in I} T_i, g \mapsto \{\lambda_i g\}_{i \in I}$$

a **random forest**.

As an example, we build a random forest with the Phython code in Appendices Section A. To get the basic idea of a random forest, we use 3 decision trees only and set the maximal depth to 3 again. We build each tree with all data points but restrict the attributes by the square root of the total number of attributes, which is the standard in the python implementation.

Example 7.1. A random forest build on the red wine data set consisting of three decision trees.

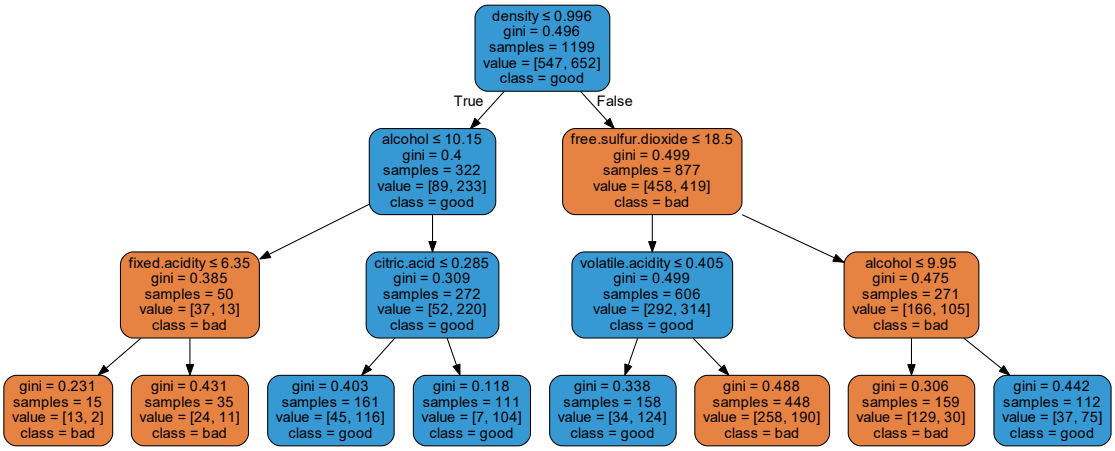


Figure 7.16: Decision tree 1 of the random forest

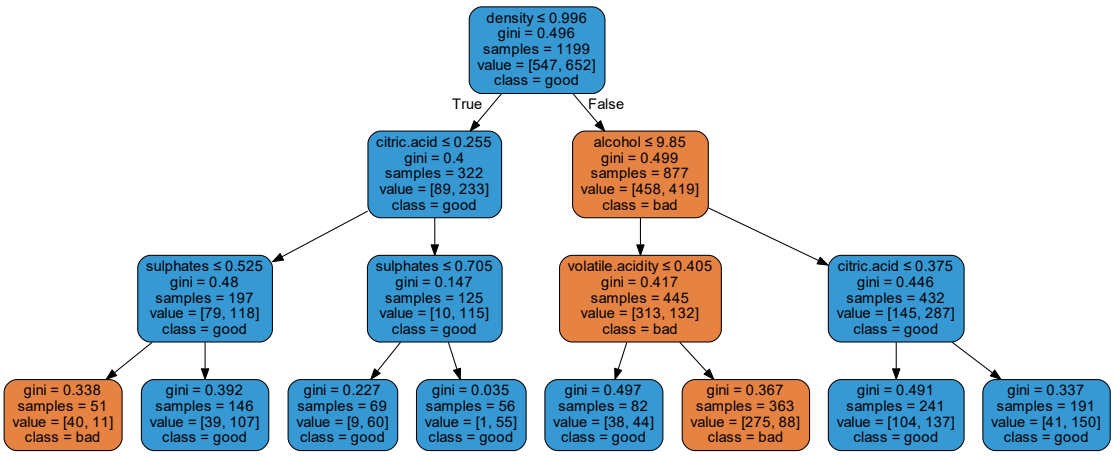


Figure 7.17: Decision tree 2 of the random forest

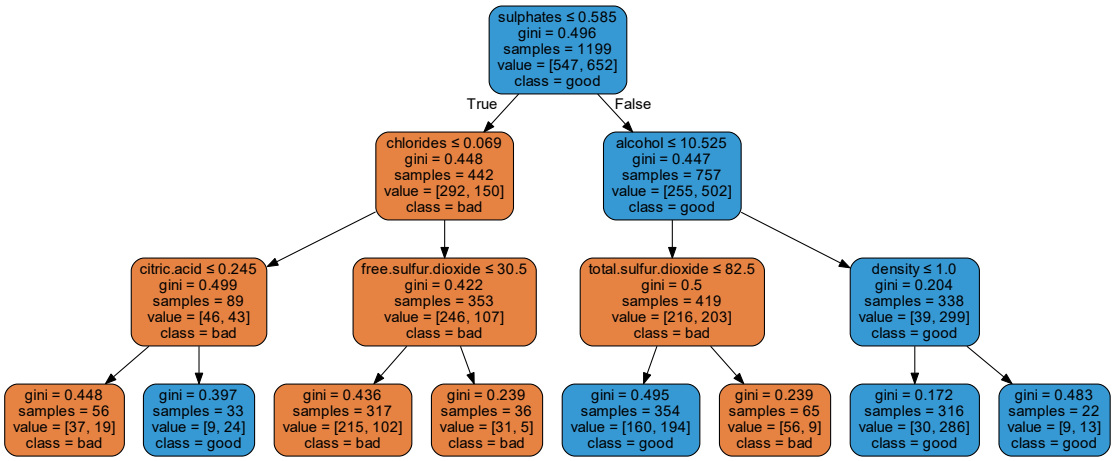


Figure 7.18: Decision tree 3 of the random forest

We use this random forest to predict the classes of our test set from the red wine data set. This leads to the following results.

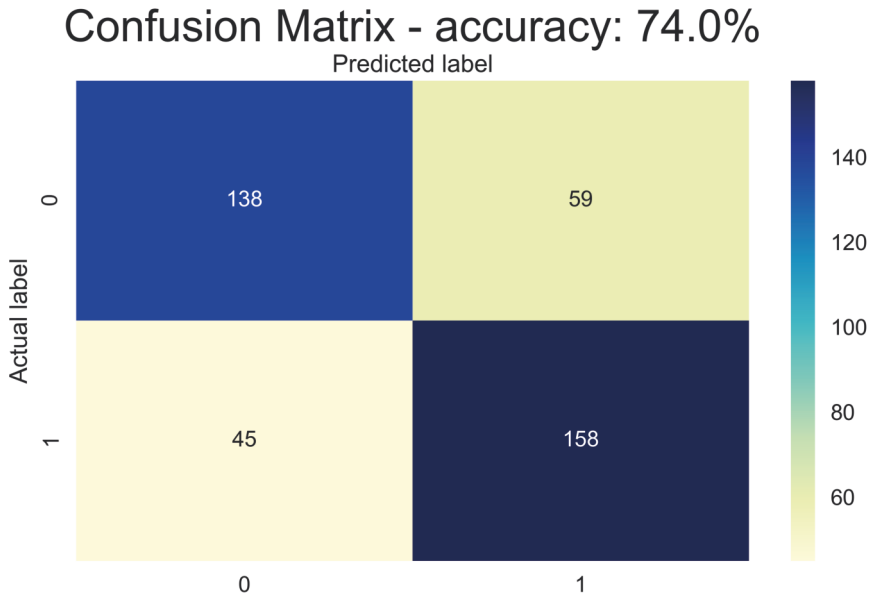


Figure 7.19: Prediction on the red wine test set using the random forest

Compared to a single decision tree, the random forest approach using 3 decision trees improve prediction accuracy by 3%. To demonstrate the potential of random forests, we construct a random forest with 100 decision trees, resulting in the following confusion matrix.

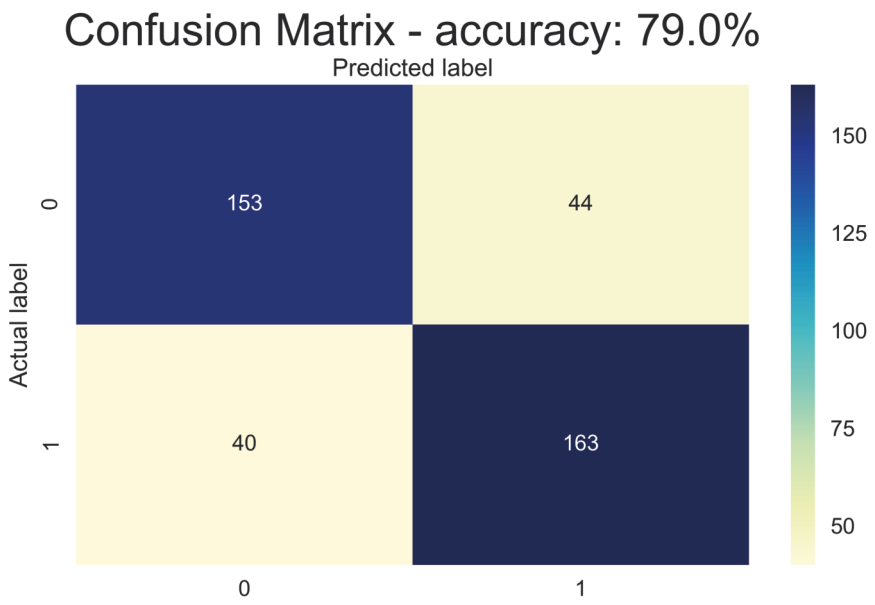


Figure 7.20: Prediction on the red wine test set using a random forest consisting of 100 trees

The last section has been an introduction into the world of decision trees and random forests. In the next section we are going to describe a connection between pattern structures and decision trees.

7.3 Making the Link

There is research dealing with the connection between formal concept analysis and decision trees (see, for example, [Kuz04, GBV+08, BBOV09]). In this section we are going to present a way to construct a pattern structure from a given decision tree. Our starting point is the following pattern setup:

Construction 7.2. Let $\mathcal{T}^s := (G, \mathbb{T}, \mathbb{T}^c, \lambda, \overline{\lambda}, \tau)$ be a decision tree setup, then

$$\mathcal{P} := (G, \mathbb{T}, \lambda)$$

is a pattern setup. We embed the pattern setup above in the embedded pattern structure

$$\mathcal{P}^e := (G, \mathbb{T}^c, \mathbb{T}, \overline{\lambda}).$$

This leads to the pattern structure $(G, \mathbb{T}^c, \overline{\lambda})$.

The construction provides a connection between decision trees and pattern structures. Below we show the link between pattern structures and random forests. For this purpose we have to look at the product of pattern structures.

Construction 7.3. Let I be a finite set and $\mathcal{P}_i := (G, \mathbb{D}_i, \delta_i)$ a pattern structure for every $i \in I$. Then,

$$\mathcal{P} := (G, \mathbb{D}, \delta)$$

with

$$D := \prod_{i \in I} D_i,\ \mathbb{D} := \prod_{i \in I} \mathbb{D}_i \text{ and } \delta : G \to \prod_{i \in I} D_i,\ g \mapsto \{\delta_i g\}_{i \in I}$$

is a pattern structure too.

Our construction shows that a product of pattern structures leads to a pattern structure again. A decision tree can be described via a pattern structure and, since a random forest is a product of decision trees, a random forest is characterized by a pattern structure too. The following construction lays this out.

Construction 7.4. Let I be an index set and $\mathcal{T}_i^s := (G, \mathbb{T}_i, \mathbb{T}_i^c, \lambda_i, \overline{\lambda_i}, \tau_i)$ a decision tree setup for all $i \in I$, and $\mathbb{F} := (G, \mathbb{T}_{prod}, \lambda)$ the corresponding random forest. Construction 7.2 provides us the embedded pattern structures $\mathcal{P}_i^e := (G, \mathbb{T}_i^c, \mathbb{T}_i, \overline{\lambda}_i)$ for every tree $\mathbb{T}_i$. Then

$$\mathcal{P} := (G, \mathbb{T}, \lambda) \text{ with}$$

$$T := \prod_{i \in I} T_i^c,\ \mathbb{T} := \prod_{i \in I} \mathbb{T}_i^c \text{ and } \lambda : G \to \prod_{i \in I} T_i^c,\ g \mapsto \{\overline{\lambda_i} g\}_{i \in I}$$

is a pattern structure too.

Construction 7.4 presents a way to describe a random forest through a pattern structure. Not only does this novel perspective enable a deeper understanding of random forests, but it also proves useful in real world applications, as will be pointed out in the next section.
Random forests are a beneficial tool, as this example shows, but it is hard to comprehend how a decision tree gives a ruling. Interval pattern structures, introduced in [KKND11], can be helpful, as the next section shows.

7.3.1 From Evaluation Map to Interval Pattern Structures

Many data mining relevant data sets (like the red wine data set) can be described by an evaluation map from Definition 2.5. For better readability, we recollect the definition: Let G be a finite set and M a set of attributes. Furthermore, let $\mathbb{W}_m := (W_m, \leqslant_m)$ be a complete lattice for every attribute $m \in M$. Further, let

$$W := \prod_{m \in M} W_m \text{ and } \mathbb{W} := \prod_{m \in M} \mathbb{W}_m.$$

Then, a map

$$\alpha : G \to W, g \mapsto \prod_{m \in M} \{\alpha_m g\} \text{ such that } \alpha_m : G \to W_m, g \mapsto w_m$$

is called evaluation map and we call $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ an evaluation setup.

Example 7.2. In the wine data set in [CCA+09], it is possible to interpret the wines as a set G, the describing attributes as the set M, and $\mathbb{W}_m$ as the numerical range of attribute m with the natural order.

In the above example, the evaluation map $\alpha : G \to W$ assigns to every wine a vector with values of all attributes $m \in M$.

Construction 7.5. Let $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ be an evaluation setup and let $\mathcal{T}^s := (G, \mathbb{T}, \mathbb{T}^c, \lambda, \overline{\lambda}, \tau)$ be a decision tree setup. Then,

$$\varphi : T^c \to \mathrm{Int}\mathbb{W}|\varphi T^c,\ t \mapsto [\inf_{\mathbb{W}} \alpha(\lambda^{-1}\tau t),\ \sup_{\mathbb{W}} \alpha(\lambda^{-1}\tau t)]$$

maps intervals to all nodes of the decision tree. The intervals are spanned by the objects in the node.

It is possible to create a pattern structure from this map, as the following theorem shows.

Theorem 7.1. Let $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ be an evaluation setup and $\mathcal{T}^s := (G, \mathbb{T}, \mathbb{T}^c, \lambda, \overline{\lambda}, \tau)$ a decision tree setup. Then

$$\mathcal{P} := (G, \mathrm{Int}\mathbb{W}|\varphi T^c, \varphi \circ \overline{\lambda}) \text{ with}$$
$$\varphi : T^c \to \mathrm{Int}\mathbb{W}|\varphi T^c,\ t \mapsto [\inf_{\mathbb{W}} \alpha(\lambda^{-1}\tau t),\ \sup_{\mathbb{W}} \alpha(\lambda^{-1}\tau t)]$$

is a pattern structure.

Proof. We want to use Application (1) of Subsection 4.3.1. Since the identity map is clearly surjective, we have to show that (id, φ) is a pattern morphism, more precisely, we have to prove that φ is a residual map. This can be achieved by applying Lemma 1.3. The preimage of a principal filter in $\mathbb{I}\mathrm{Int}\mathbb{W}|\varphi T$ under φ is always a principal filter in $\mathbb{T}^c$. For every $x \in \mathrm{Int}\mathbb{W}|\varphi T$,

$$\varphi^{-1}\{s \in \mathrm{Int}\mathbb{W}|\varphi T \mid x \leqslant s\} = \{t \in T^c \mid \varphi^{-1}x \leqslant t\}$$

holds. □

To lift the previous theorem up to a random forest, we use Construction 7.3.

Construction 7.6. Let $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ be an evaluation setup, I an index set, and $\mathbb{F} := \coprod_{i \in I} \mathbb{T}_i$ a random forest with corresponding decision tree setups $\mathcal{T}_i^s := (G, \mathbb{T}_i, \mathbb{T}_i^c, \lambda_i, \overline{\lambda_i}, \tau_i)$. Then,

$$\mathcal{P}_i := (G, \mathrm{Int}\mathbb{W}|\varphi_i T_i^c, \varphi_i \circ \overline{\lambda_i}) \text{ with}$$
$$\varphi_i : T_i^c \to \mathrm{Int}\mathbb{W}|\varphi_i T_i^c,\ t_i \mapsto [\inf_{\mathbb{W}} \alpha(\lambda_i^{-1}\tau_i t_i),\ \sup_{\mathbb{W}} \alpha(\lambda_i^{-1}\tau_i t_i)]$$

is a pattern structure and

$$\mathcal{P} := (G, \mathbb{D}, \delta)$$

with

$$D := \prod_{i\in I} \mathrm{Int}\mathbb{W}|\varphi_i T_i^c,\ \mathbb{D} := \prod_{i\in I} \mathbb{I}\mathrm{nt}\mathbb{W}|\varphi_i T_i^c \text{ and } \delta : G \to \prod_{i\in I} \mathrm{Int}\mathbb{W}|\varphi_i T_i^c,\ g \mapsto \{\varphi_i \circ \overline{\lambda}_i g\}_{i\in I}$$

is a pattern structure, too.

7.4 Real World Application

To get familiar with the constructions from the last section and show that this is a useful view on things, we will present an example in the following. Starting point is a random forest with 10 decision trees on the red wine data set of Section 7.1. The random forest was built on the training set and delivers the result below by predicting the classes of the test set.

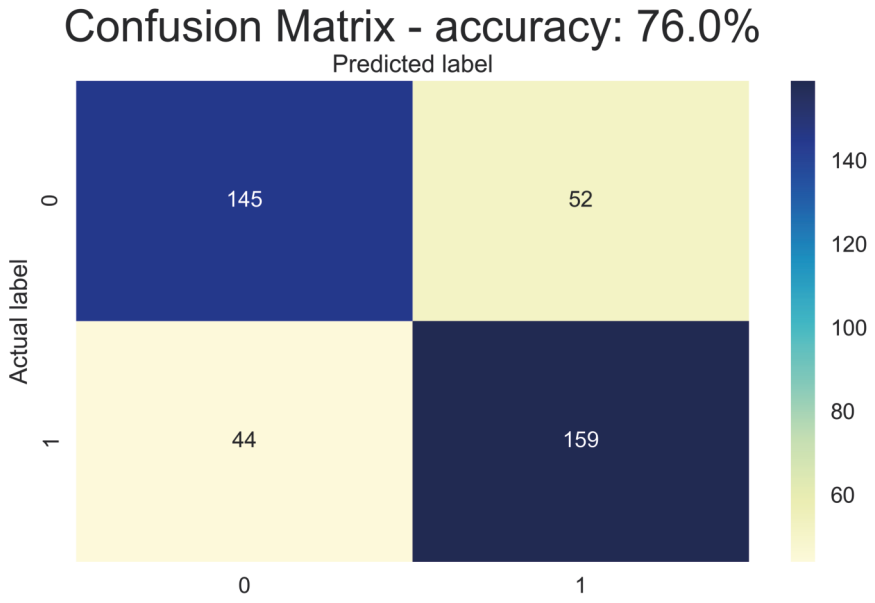


Figure 7.21: Predicting the classes of the test set

If we consider the wines of the red wine data set as a set G, then Construction 7.6 provides a set of 10 intervals for every $g \in G$, build of the 10 leaves which contain g. That is because g contains in exactly one leaf in every of the 10 decision trees. The intersection of these intervals can not be empty because g is at least in every interval. A predicted class probability is assigned to every g by a decision tree, which can be interpreted as a purity measure, calculated as the ratio of good examples to all examples in a leaf. For a random forest, the average of this ratio for all leaves, which contains g is used to calculate a score. We took the best scored g of our training set and looked at the union of the intervals constructed by the random forest. For better readability, we scaled the attributes linear into the interval $[0,1]$ by simply setting the largest value of each attribute to 1 and the smallest to 0. We received the following result:

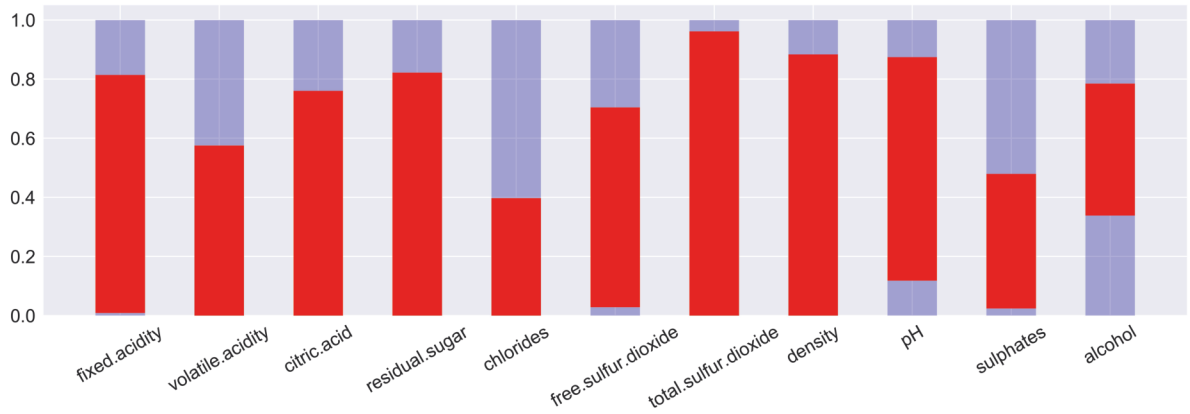


Figure 7.22: Interval built from the best scored wine (features scaled in the range $[0, 1]$)

This pattern contains 430 wines of the training set, 355 of which were rated good and 75 were rated bad. So, there is a 82,6% chance that a wine of our training set, which lies in this interval, is a good wine. Also on the test set this is an useful interval. If we use it to predict the classes of the test set, we receive the following confusion matrix:

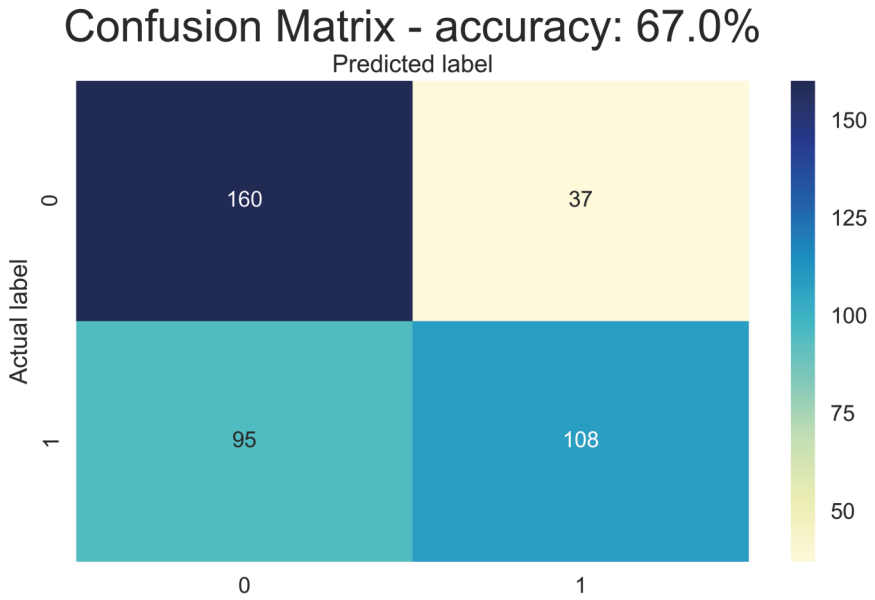


Figure 7.23: Predicting the classes of the test set with Interval 1 (Figure 7.22)

As a reading example: there are 145 (108+37) wines of the test set in this interval. 108 of them are good. So, a wine of the test set, which lies in this interval is with a chance of 74,5% a good wine. This best interval delivers some interesting insights into the data set. However, it is also fascinating to look at the pattern of the worst scored wine, which is shown in Figure 7.24.

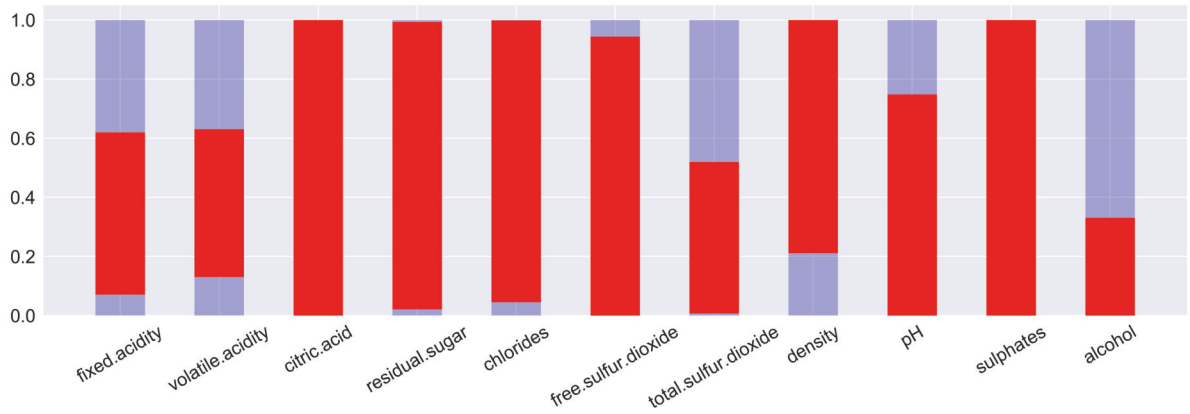


Figure 7.24: Interval built from the worst scored wine (features scaled in the range $[0,1]$)

This pattern includes 671 wines, 235 of which are good. This means that a wine of the test set, which lies in this interval is with a probability of 65,0% a bad wine. In comparison to the interval of the best scored wine, the different range in the attribute alcohol is significant. Indeed, the average alcohol content of the good wines (10,86%) in the data set is almost 1% higher than the alcohol content of the bad wines (9,93%). As mentioned before, the training set contains 75% of the data, which are 1199 wines. 652 of them are good and 547 bad. The interval of the best scored wine (Figure 7.22) only includes 430 wines. To choose a collection of intervals for a prediction model, we calculate a score on a pattern p, as mentioned before, via

$$score(p) = \frac{good\ wines\ in\ p}{all\ good\ wines}.$$

To improve interpretability, we select only the following intervals from the 5 best scored wines to build our model. The interval of the best wine is already known from Figure 7.22 but, for better readability, we printed it here too. We looked at the union of the intervals and scaled the range of the features to $[0,1]$ again.

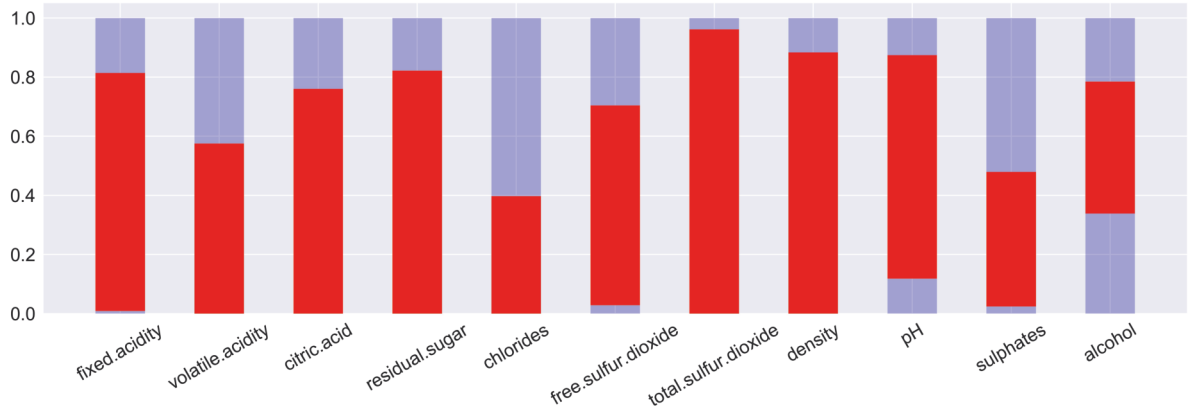


Figure 7.25: Interval 1 contains altogether 430 wines, 355 good and 75 bad

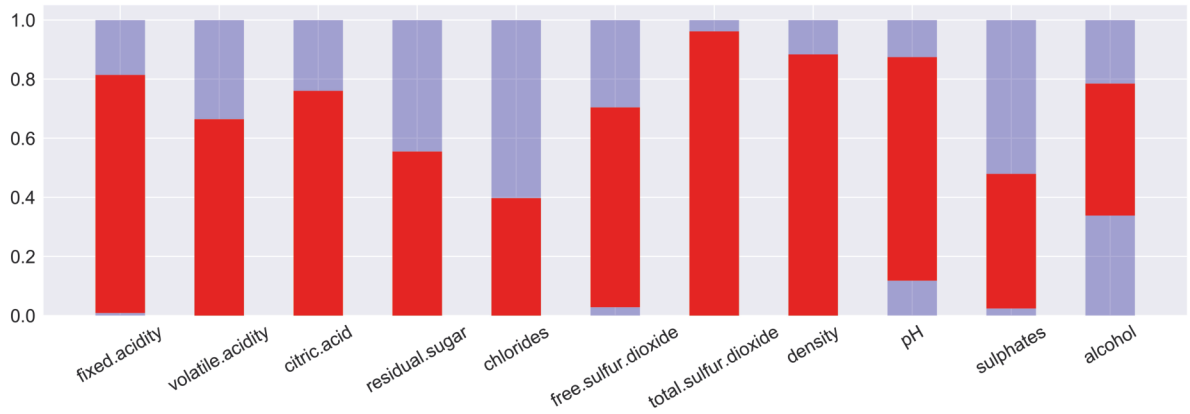


Figure 7.26: Interval 2 contains altogether 435 wines, 357 good and 78 bad

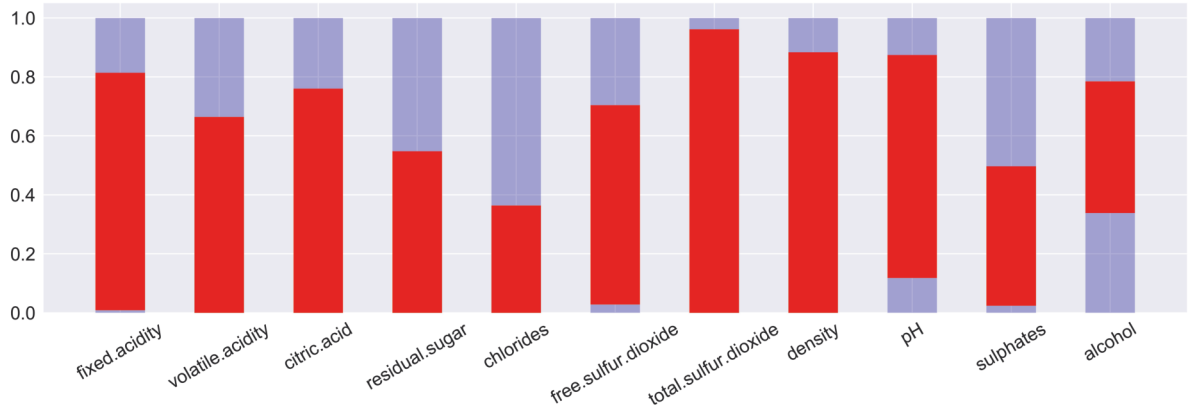


Figure 7.27: Interval 3 contains altogether 434 wines, 356 good and 78 bad

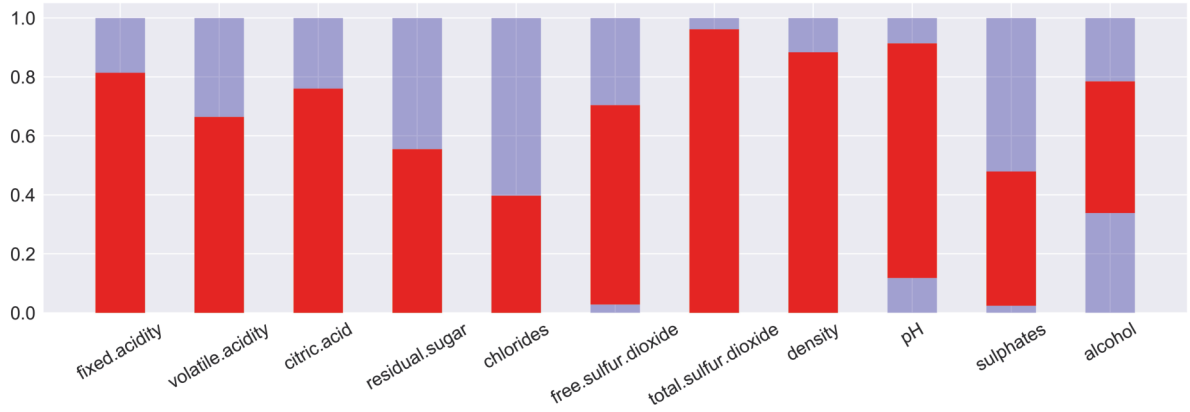


Figure 7.28: Interval 4 contains altogether 437 wines, 358 good and 79 bad

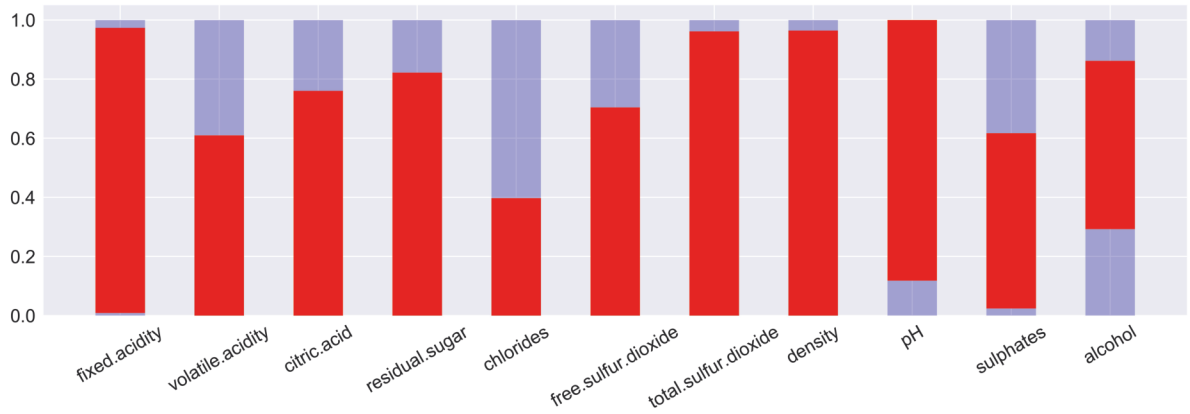


Figure 7.29: Interval 5 contains altogether 552 wines, 426 good and 126 bad

Combining these 5 intervals to predict the classes of the test set leads to the following confusion matrix:

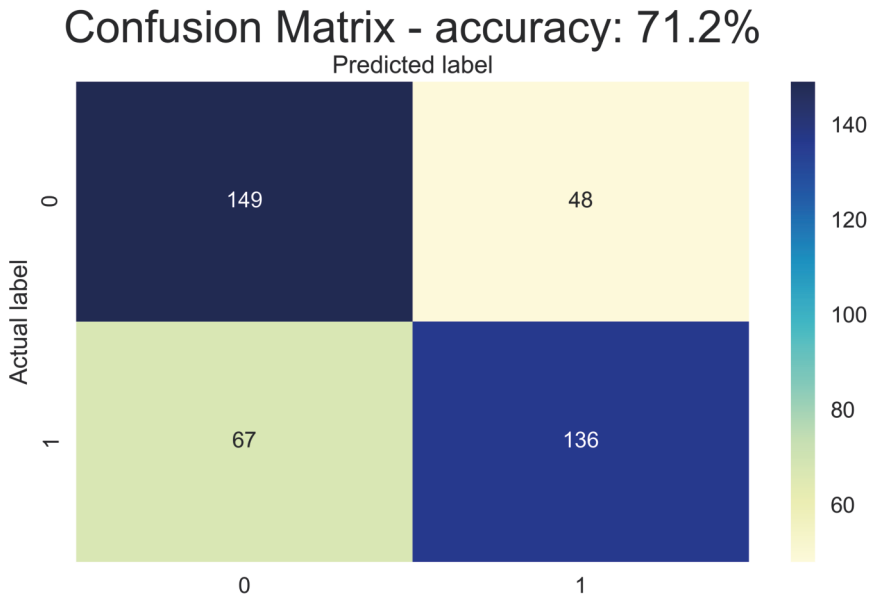


Figure 7.30: Combination of the 5 intervals

Prediction accuracy decreased somewhat, but the high interpretability makes our algorithm useful. For example, density, which is the eighth attribute, seems unimportant for the quality of a wine since all patterns have a huge range in this feature. On the other hand, the range in the attribute sulphates is comparatively small. This suggests that sulphates play an important role in classifying a wine of the data set.

7.5 Conclusion and Critical Discussion

We introduced a novel framework for the application of pattern structures. The previous chapter presented a new way of building a pattern structure through a given decision tree. This link is an interesting point of view and via a visualisation it leads to a better understanding of how a decision tree gives a ruling. Furthermore, this chapter shows that the product of pattern structures are also pattern structures and so we extend the link between pattern structures and decision trees to random forests since they are a product of decision trees. We also introduced a model to predict classes of red wines. This model is an abstraction of a random forest. As introduced here, the prediction of the random forest is better, but the high interpretability makes our algorithm valuable. This is just a first useful example of our method. Further investigations on other data sets have to prove the importance of our algorithm. Our approach could maybe be improved. For example, the intervals in Figure 7.26, Figure 7.27 and Figure 7.28 look similar. Maybe there is a better way to select the intervals for the predicting model?
Another point to improve: with an increasing number of decision trees in the random forest the range of the intervals increases. The following figures show the intervals of the best scored wine of three random forest algorithms (2, 10 and 50 decision trees). Appendices Subsection B.1 contains the random forest confusion matrix, the intervals of the best scored wine, and a test set prediction for each case. The first interval (Figure 7.31) contains 108 wines of the training set, 99 of which were rated as good. Hence, there is a chance of 91,7% that a wine of the test set, which lies in this interval, is a good wine. Figure 7.32 shows an interval, which contains 430 wines of the training set, 355 of which are good. As a last example, we choose a random forest with 50 decision trees. The resulting interval contains 1075 wines, 603 of which are good. Considering that the training set contains 1199 wines altogether, it is obvious that the range of this interval is too large and unusable for a prediction model, but for completness the interval is shown in Figure 7.33.

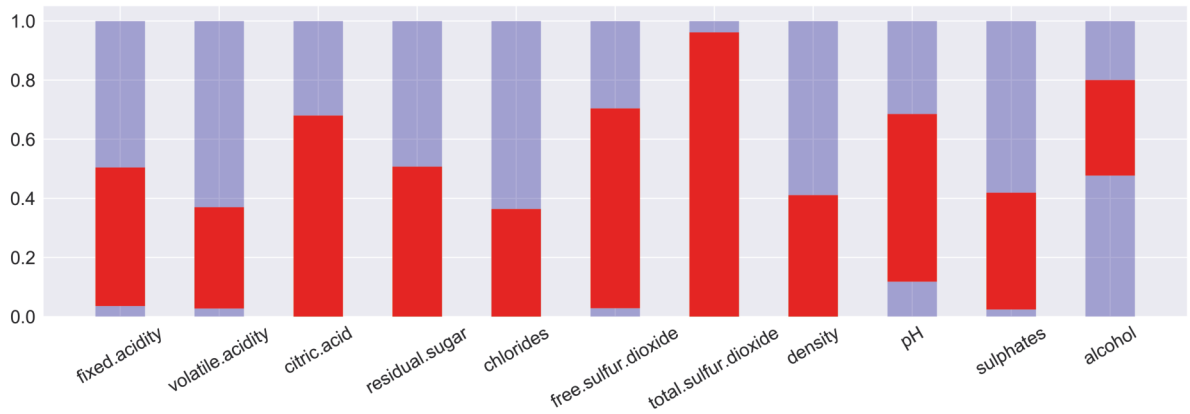


Figure 7.31: Interval built from 2 decision trees (features scaled in the range $[0, 1]$)

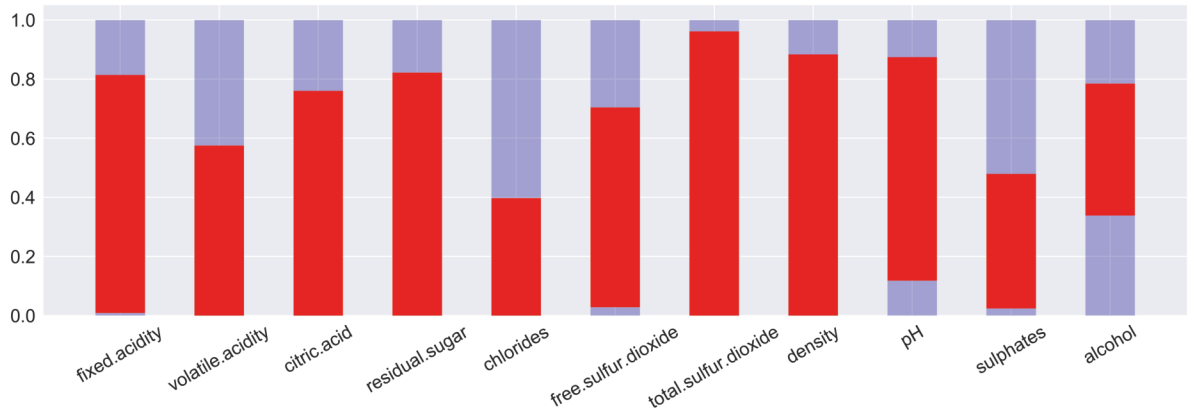


Figure 7.32: Interval built from 10 decision trees (features scaled in the range $[0, 1]$)

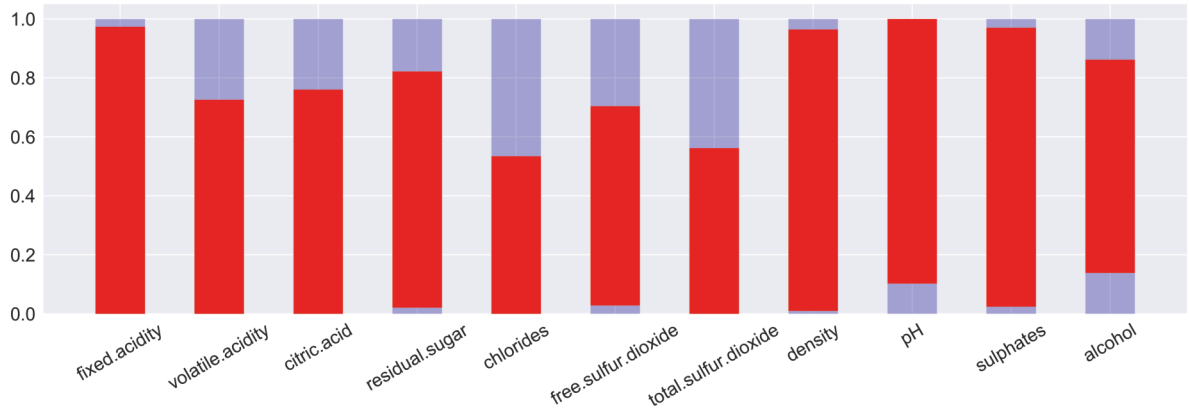


Figure 7.33: Interval built from 50 decision trees (features scaled in the range $[0, 1]$)

As a matter of fact we point out, that an increasing amount of decision trees in a random forest in general increases the quality of the prediction of the random forest, but unfortunately decreases the prediction quality of our method. Further investigations are needed to solve this problem.

8 Patterns via Clustering

In this section we are going to present another example to create useful patterns. We describe our proceeding on the red wine data set from Section 7.1. First we give a general framework for creating a pattern structure. Parts of this chapter and a similar example were presented at the Workshop "What can FCA do for Artificial Intelligence?" (FCA4AI, 2020) [LS20b].

8.1 From an Elementary Pattern Structure to an Interval Pattern Structure

We start with an evaluation setup $\mathcal{E} := (G, M, \mathbb{W}, \alpha)$ introduced in Definition 2.5. For recollection: in that case G is a finite set and M a set of attributes. Furthermore, $\mathbb{W}_m := (W_m, \leqslant_m)$ is a complete lattice for every attribute $m \in M$ and

$$W := \prod_{m \in M} W_m \text{ and } \mathbb{W} := \prod_{m \in M} \mathbb{W}_m.$$

The corresponding evaluation map to an evaluation setup is given by

$$\alpha : G \to W, g \mapsto \prod_{m \in M} \{\alpha_m g\}$$

such that

$$\alpha_m : G \to W_m, g \mapsto w_m.$$

Such an evaluation map is a good starting point for an elementary pattern structure from Definition 2.1, with

$$L := \prod_{m \in M} W_m \text{ and } \mathbb{L} := \prod_{m \in M} \mathbb{W}_m.$$

This leads to the elementary pattern structure given by

$$(G, \mathbb{E}, \varepsilon) \text{ with } \mathbb{E} := (2^L, \supseteq)$$

where

$$\varepsilon : G \to 2^L, g \mapsto \{\alpha g\}.$$

Thus, $\mathbb{E} := (2^L, \supseteq)$ is the dually ordered power set of vectors with values of the attributes, which describe the objects $g \in G$. On $\mathbb{E}$ we installed the following kernel operator

$$\gamma : 2^L \to 2^L, X \mapsto [inf_{\mathbb{L}} X, sup_{\mathbb{L}} X]_{\mathbb{L}}.$$

This leads to the o-projection of the elementary pattern structure $\mathcal{P}_\rho$ via γ, that is,

$$(G, \mathbb{D}, \delta) := opr(\mathcal{P}_\rho, \gamma).$$

As a matter of fact,

$$\mathbb{D} = (D, \supseteq) \text{ with } D = \text{Int}\mathbb{L}$$

is the dual interval lattice of $\mathbb{L}$, and the map δ is given by

$$\delta : G \to D,\ g \mapsto \{\alpha g\}.$$

The following example deals with this construction more closely.

Example 8.1. Similarly to Example 7.2, we can interpret the wines from the wine data set introduced in Section 7.1 as a set G, the describing attributes as the set M and $\mathbb{W}_m$ as the numerical range of attribute m with the natural order. In this example the evaluation map

$$\alpha : G \to W$$

assigns to every wine a vector with values of all attributes $m \in M$. So, the patterns in $\mathbb{E}$ consist of all possible subsets of these vectors. Via the kernel operator on $\mathbb{E}$

$$\gamma : 2^L \to 2^L, X \mapsto [inf_{\mathbb{L}} X, sup_{\mathbb{L}} X]_{\mathbb{L}}$$

we get patterns in $\mathbb{D}$ of the patterns in $\mathbb{E}$. Since γ is a closure operator on the power set $\mathbf{2}^L := (2^L, \subseteq)$ we can think of the patterns in $\mathbb{D}$ as closures of patterns in $\mathbb{E}$.

Often the dual power set lattice $\mathbb{E}$ is too large for applications. Therefore, we concentrate on relevant patterns in $\mathbb{D}$, that is, in the dual interval lattice of $\mathbb{L}$. In the next section we apply our approach to the red wine data set of Section 7.1. More percisely, as an example of the findings of this section, we describe how we predict classes of red wines.

8.2 Mining for Relevant Patterns

We use the red wine data set from Section 7.1 and split it into a training set (75% of the data) and a test set (25% of the data) as described in Section 7.1. We use the training set to build the pattern structure and then use it to predict the classes of the wines in the test set. To identify important patterns in $\mathbb{E}$ for the red wine classification, we looked at the positive examples of the training set and combined the results of different clustering algorithms implemented in Python. In particular, we used a k-means and k-medoids algorithm with metrices Mahalanobis, Euclidean and correlation of Section 1.3. Furthermore, we interpret the leaves of decision trees (with gini impurity and entropy as splitting measures) as clusters of wines to find important patterns in $\mathbb{E}$ for our case. The same clustering algorithm can lead to different output clusters, this resulting from the different metrics used to measure the distance and the randomly choosen starting points of the algorithms. Therefore, we ran every algorithm 10 times using different specifications for every attempt. The number of clusters for the k-medoids and k-means algorithms is set randomly between 10 and 150. For the decision trees we set the number of examples in a leaf to at least 100. This leads to more than 700 clusters in $\mathbb{E}$.
Via the kernel operator on $\mathbb{E}$:

$$\gamma : 2^L \to 2^L, X \mapsto [inf_{\mathbb{L}} X, sup_{\mathbb{L}} X]_{\mathbb{L}},$$

we get patterns in $\mathbb{D}$ of the clusters in $\mathbb{E}$. Since γ is a closure operator on the power set $\mathbf{2}^L := (2^L, \subseteq)$, we can think of the patterns as closures of clusters.
As the next step, we eliminated all clusters with less than 100 wines. Then we looked at the ratio of good examples (wines with a scoring of 5 or better) and all examples (good and bad) in the patterns and took the five patterns with the best ratio. These patterns are listed below. We used the Python code under Appendices Section C to create them. For better interpretability, we scaled every attribute linear to the range $[0, 1]$.

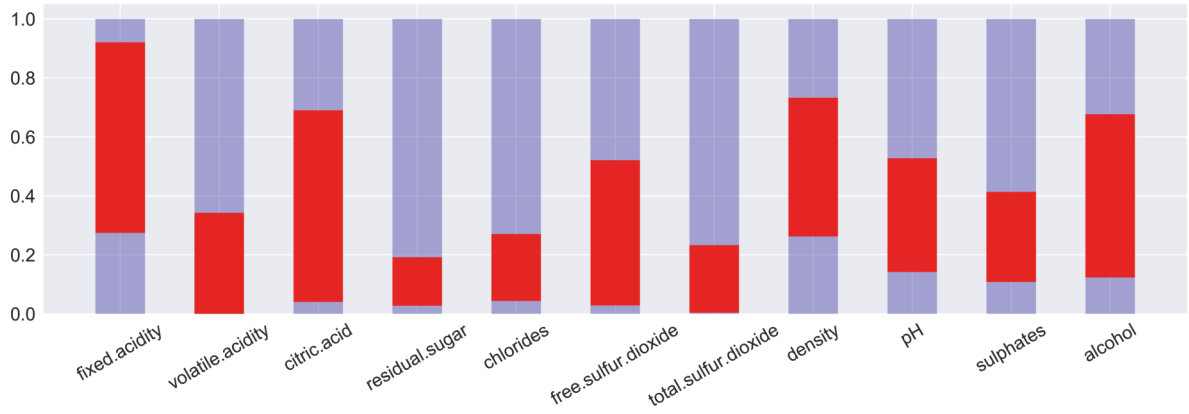


Figure 8.1: Interval 1 (k-medoids - mahalanobis): 142 wines, 142 good and 0 bad

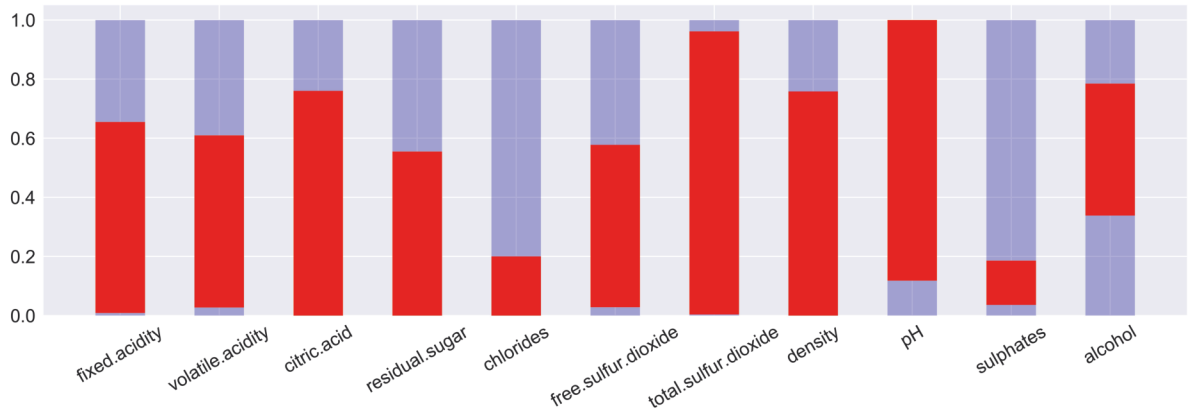


Figure 8.2: Interval 2 (decision tree - gini): 140 wines, 140 good and 0 bad

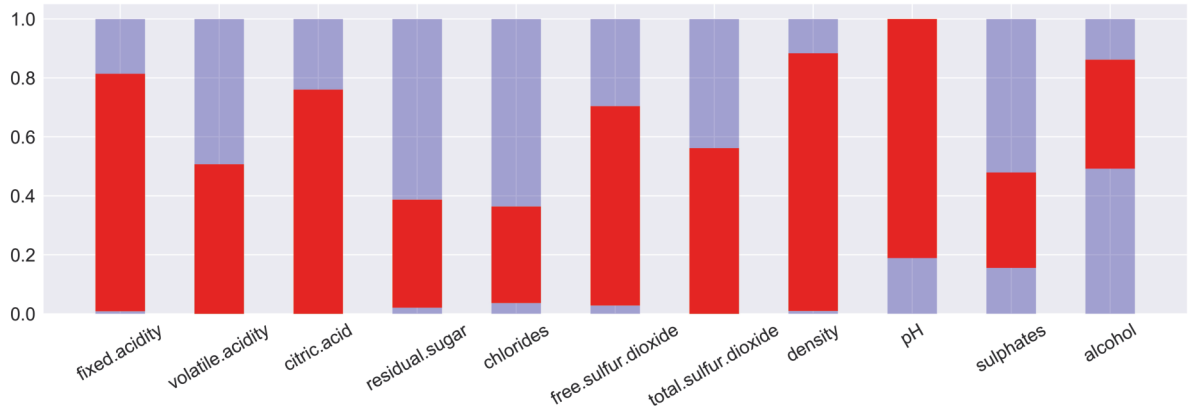


Figure 8.3: Interval 3 (decision tree - entropy): 133 wines, 133 good and 0 bad

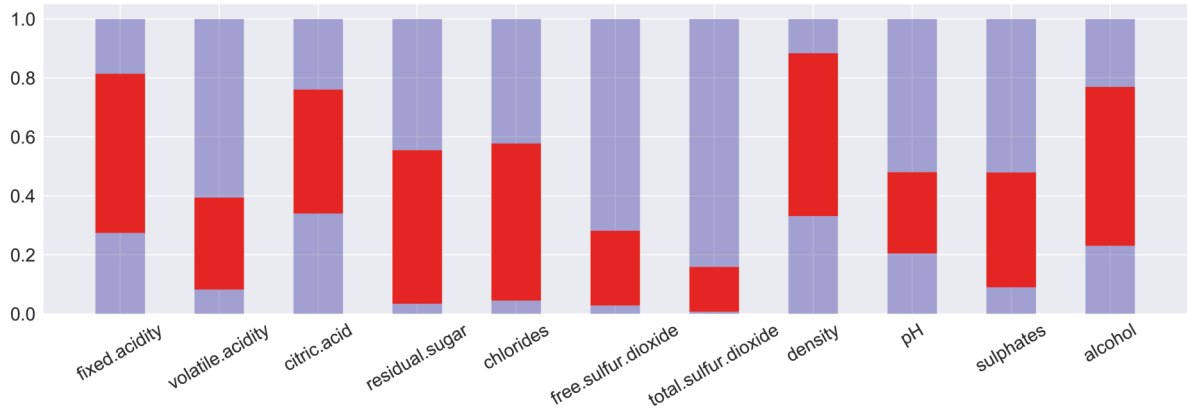


Figure 8.4: Interval 4 (k-means): 131 wines, 131 good and 0 bad

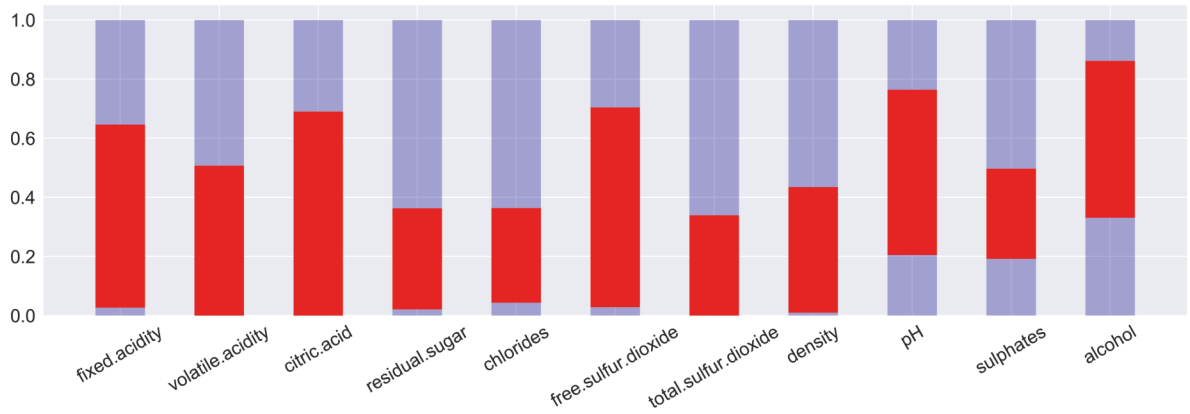


Figure 8.5: Interval 5 (decision tree - gini): 123 wines, 123 good and 0 bad

The first interval in Figure 8.1 includes 142 good wines of the test set and performs well on the test set as can be seen in the following confusion matrix.

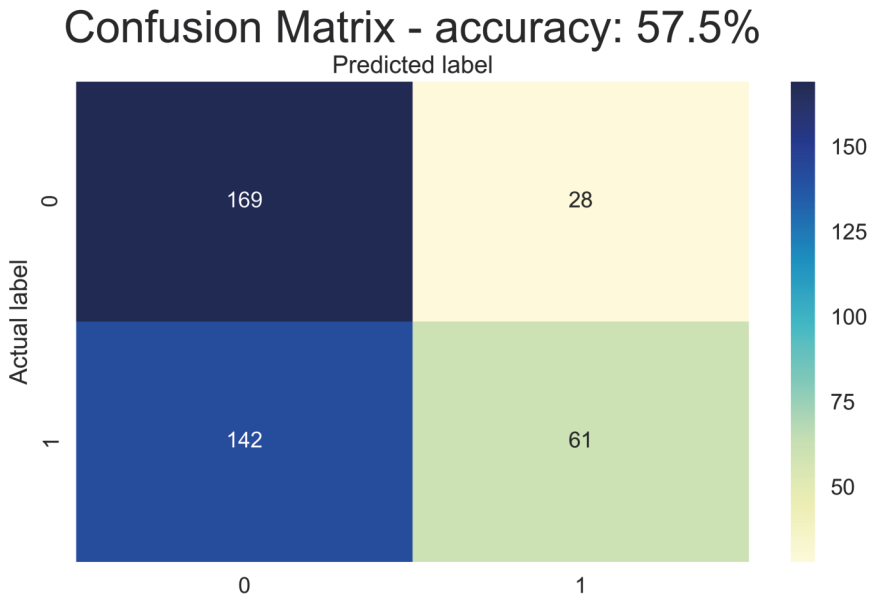


Figure 8.6: Predicting the test set with Interval 1

The Figure shows that 89 wines, 61 of which are good, of the test set lie in Interval 1 from Figure 8.1. If we combine the 5 intervals to predict the classes of the test set accuracy is further improved:

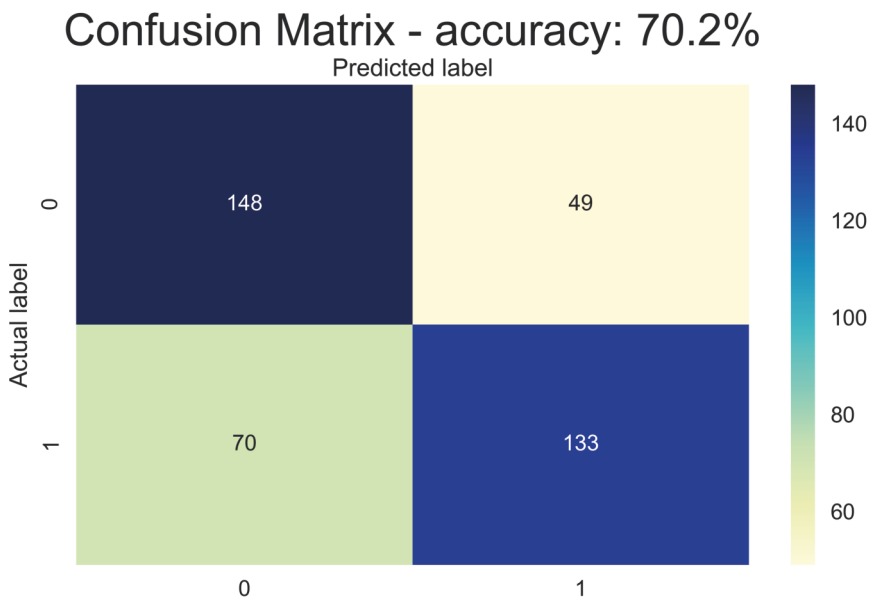


Figure 8.7: Predicting the test set with the 5 intervals

We now have 182 wines of the test set lie in one of the 5 intervals and 133 of them are good wines.

For example in [SS14, BKN15a, BKN17a, BKN17b] and in various parts of this thesis complexity reduction of the pattern space is of crucial interest. In the example presented in this section, the question also arose how a reduction on the set of patterns would impact the quality of our model. For that reason we look at every single attribute and build a model with it, using the same procedure as described earlier in this chapter. This leads to the following results:

Attribute	**Accuracy trainings set**	**Accuracy test set**
alcohol	59,9%	62,3%
volatile acidity	58,6%	58,2%
sulphates	56,6%	57,2%
chlorides	56,5%	56,0%
citric acid	51,2%	52,8%

Table 8.1: Accuracy of the best interval of the single attributes

Attributes, which are not contained in the table delivered no cluster satisfying our conditions. Alcohol is by far the most valuable attribute, that is why we try to build a model just on the feature alcohol. We receive the following intervals. On the left side is the range of the interval

and on the right side is the confusion matrix we get, when we try to predict the test set with the interval. The caption shows, how many wines of the training set are included in the interval.

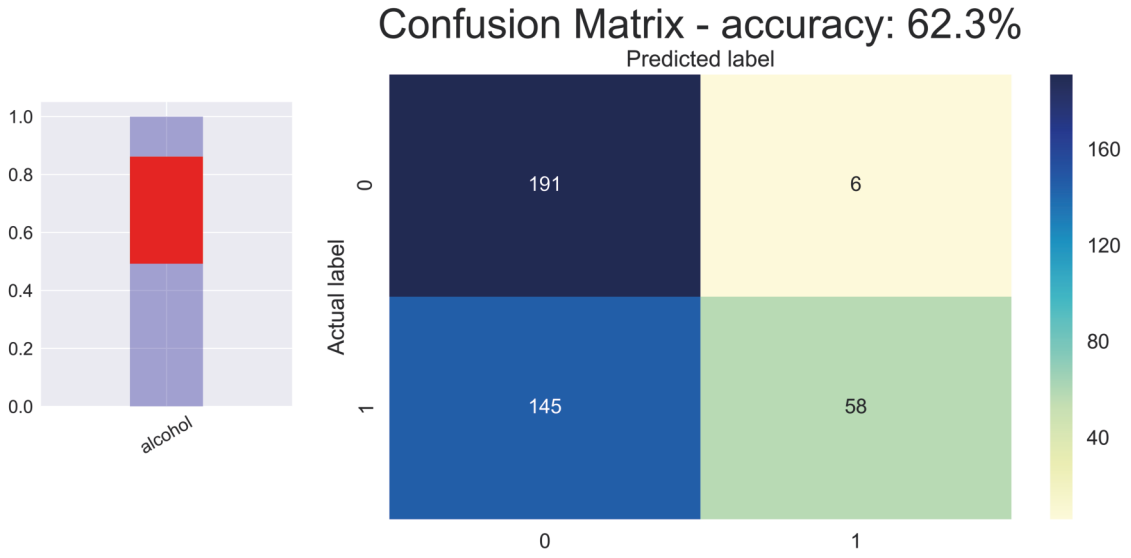


Figure 8.8: Interval 1 (decision tree - entropy): 171 wines, 171 good and 0 bad

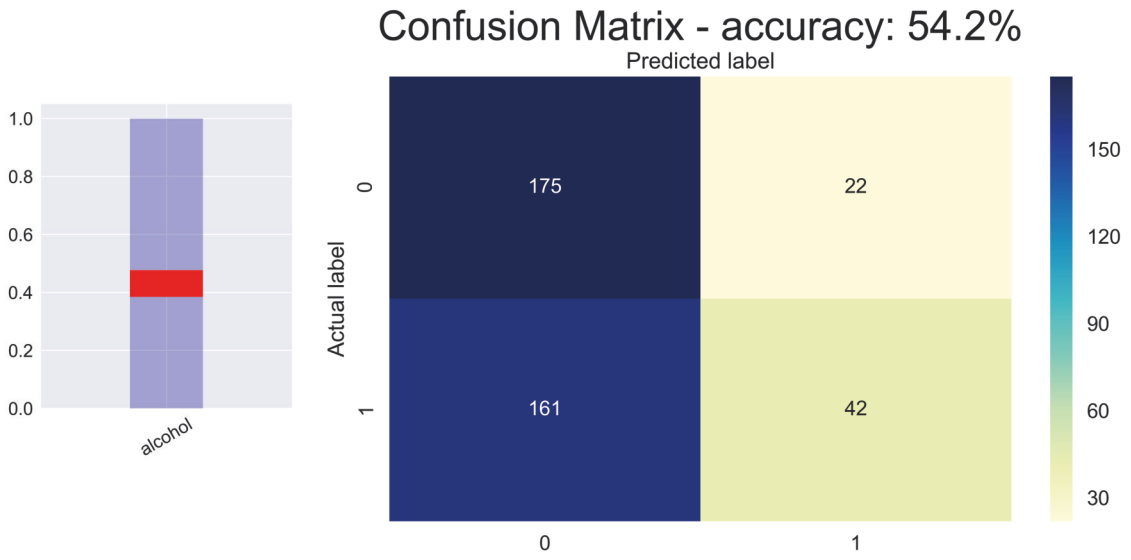


Figure 8.9: Interval 1 (k-medoids - mahalanobis): 142 wines, 142 good and 0 bad

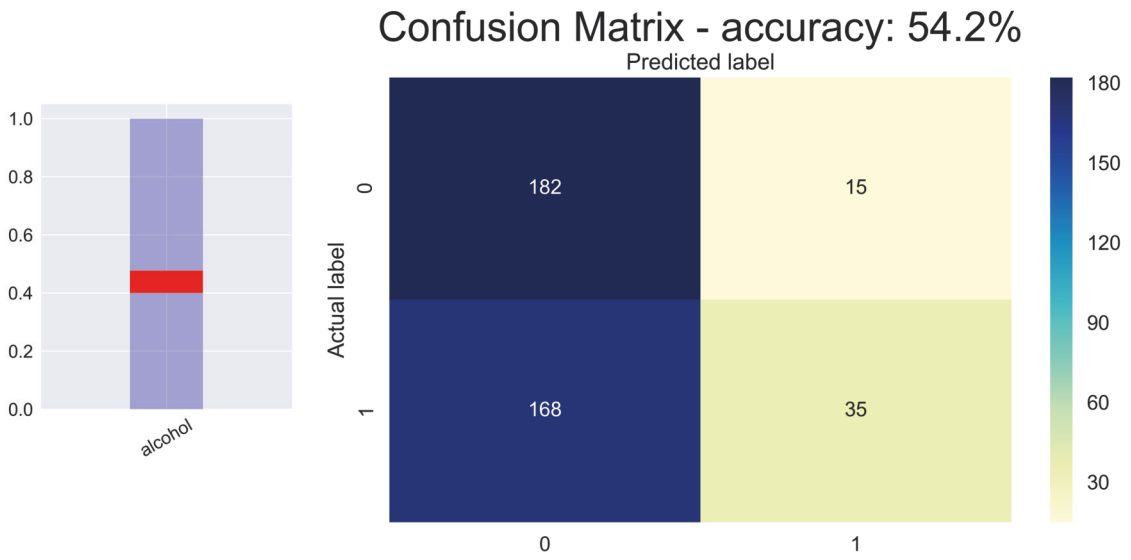


Figure 8.10: Interval 3 (decision tree - entropy): 117 wines, 117 good and 0 bad

Combining these 3 intervals to predict the test set we receive the following result.

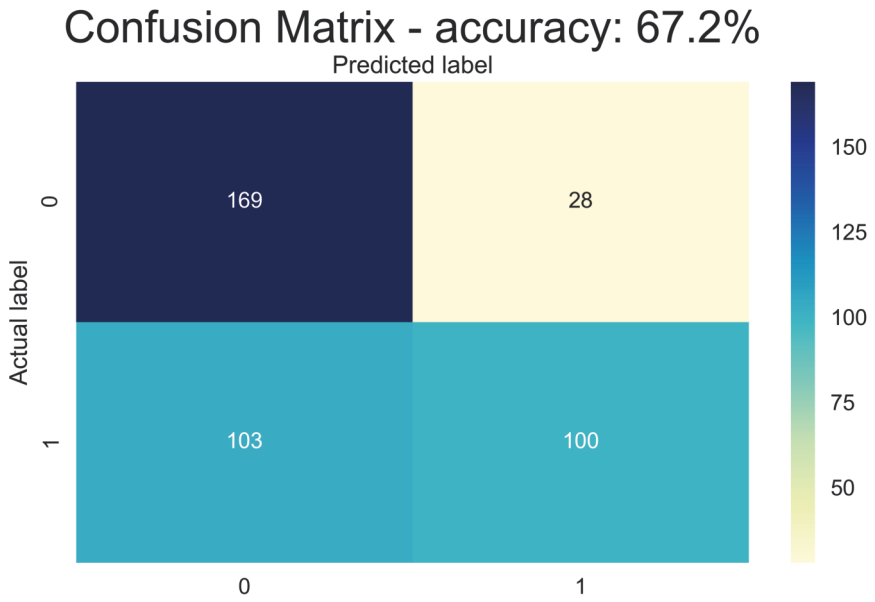


Figure 8.11: Predicting the test set with the 3 intervals

The intervals gained from the other attributes are printed in Subsection D.1. The confusion matrix shows an impressive outcome for a single attribute and the results get even better if

we consider two instead of one attribute. In the following we add attributes till the prediction quality no longer increases. We start with volatile acidity, since it delivers the second best result as stand alone attribute on the training set. Considering these two attributes we get the following intervals.

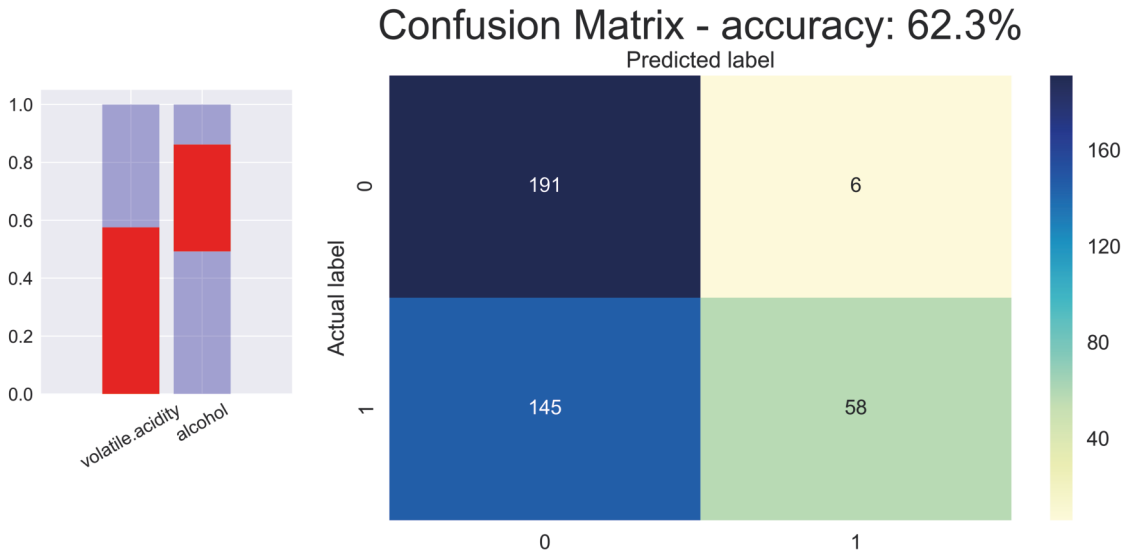


Figure 8.12: Interval 1 (decision tree - gini): 133 wines, 133 good and 0 bad

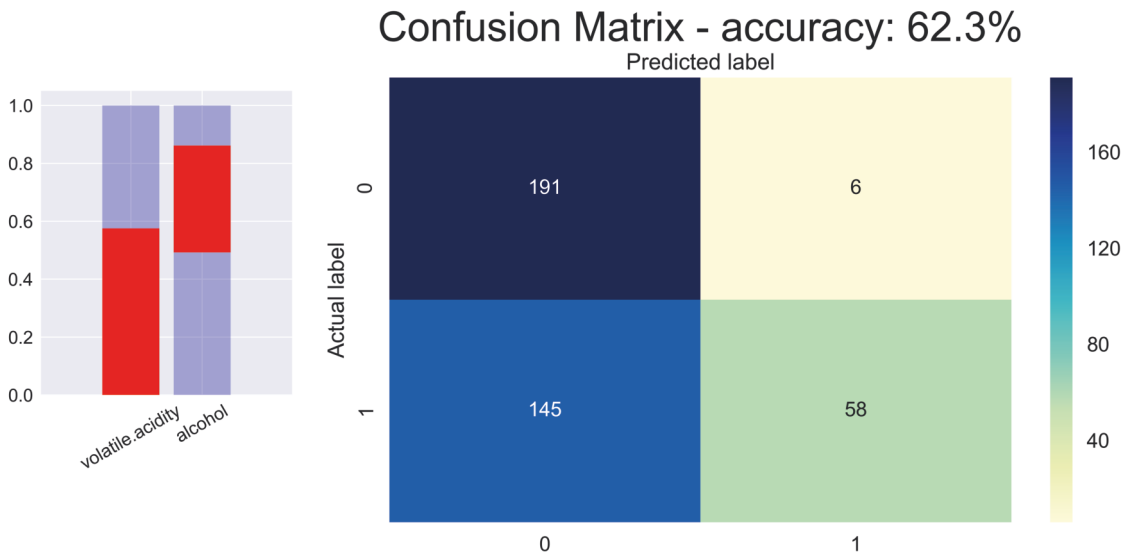


Figure 8.13: Interval 2 (decision tree - gini): 137 wines, 135 good and 2 bad

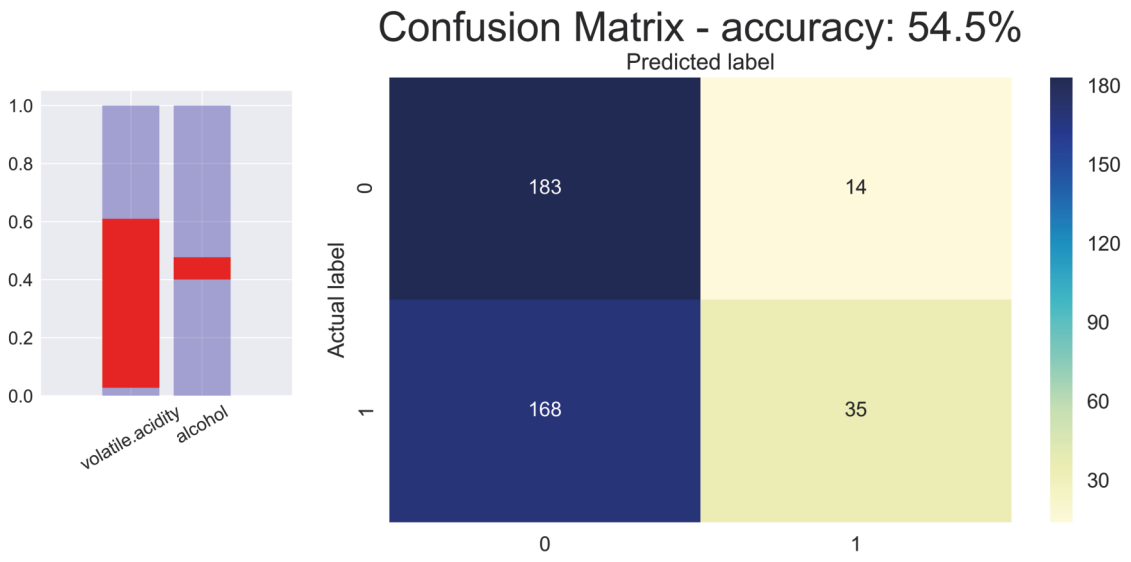


Figure 8.14: Interval 3 (decision tree - gini): 138 wines, 130 good and 8 bad

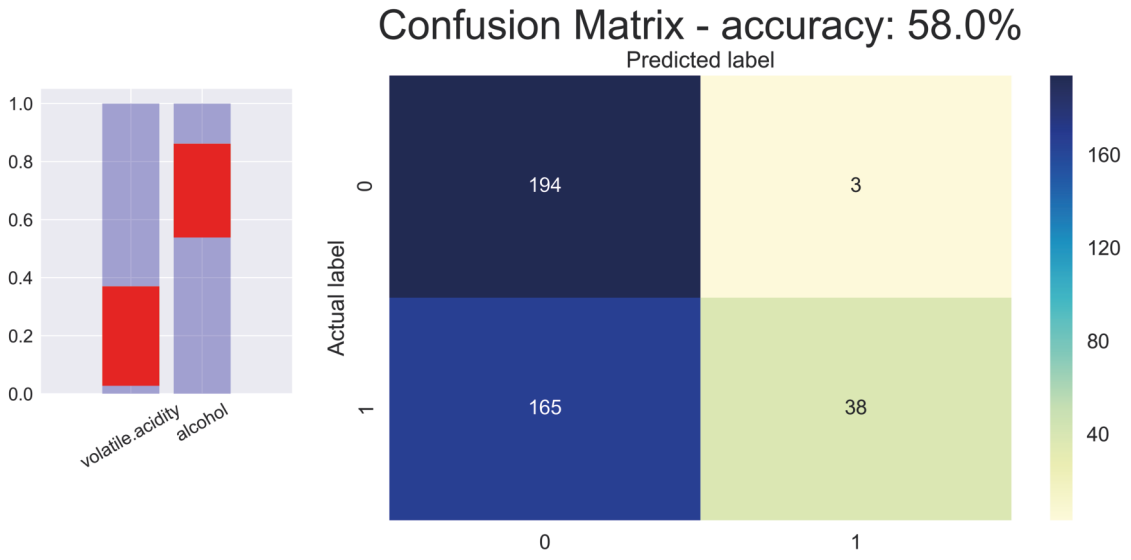


Figure 8.15: Interval 4 (k-medoids - euclidean): 120 wines, 111 good and 9 bad

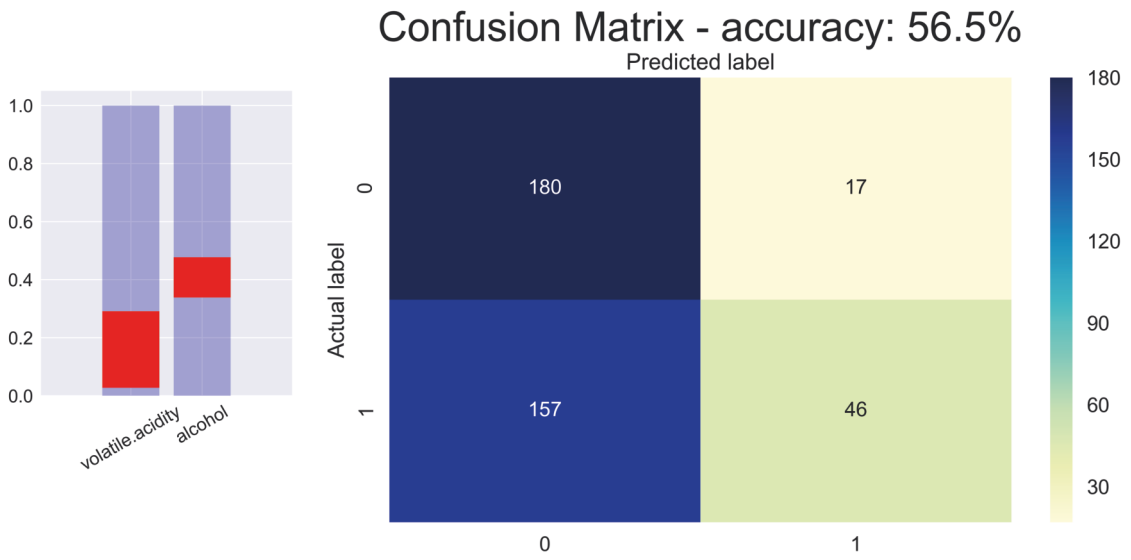


Figure 8.16: Interval 5 (k-medoids - mahalanobis): 139 wines, 121 good and 18 bad

Combining these 5 intervals to predict the test set we receive the following result.

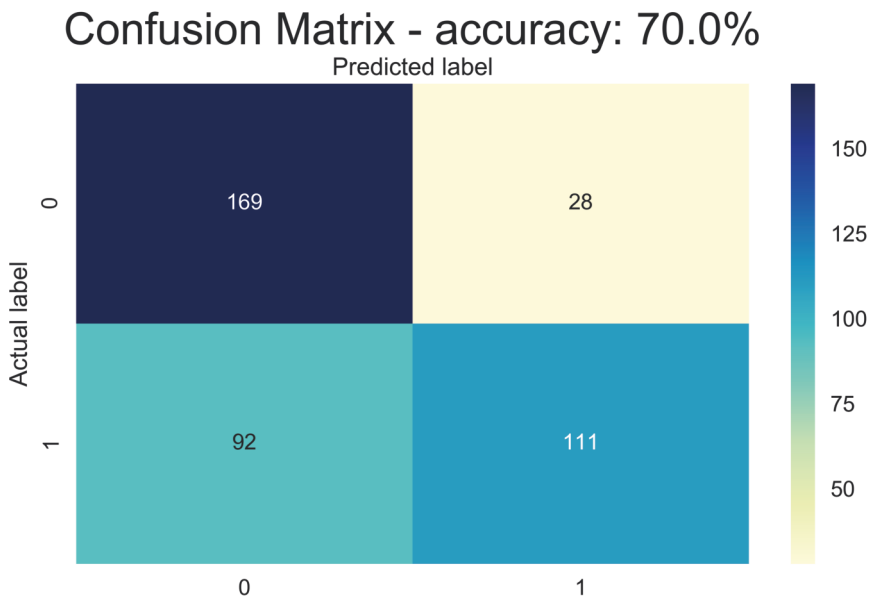


Figure 8.17: Predicting the test set with the 5 intervals

It makes sense to add the attribute sulphates too, as we can see in the next intervals, where this attribute is included. We choose this attribute because it has the third best performance as stand alone attribute and the prediction quality on the training set increases if we include the

attribute sulphates.

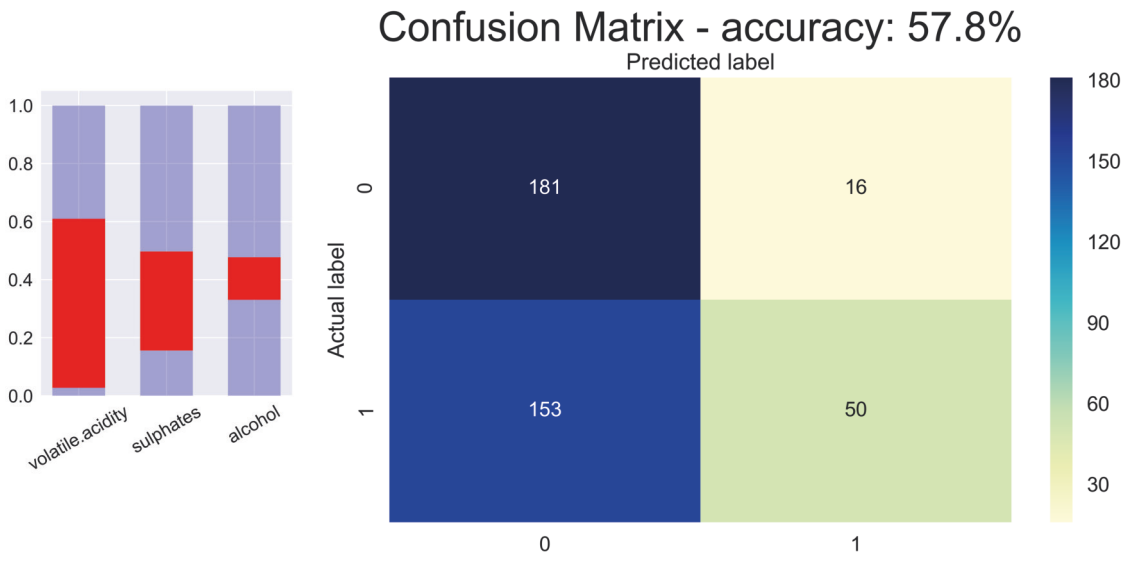


Figure 8.18: Interval 1 (decision tree - gini): 166 wines, 166 good and 0 bad

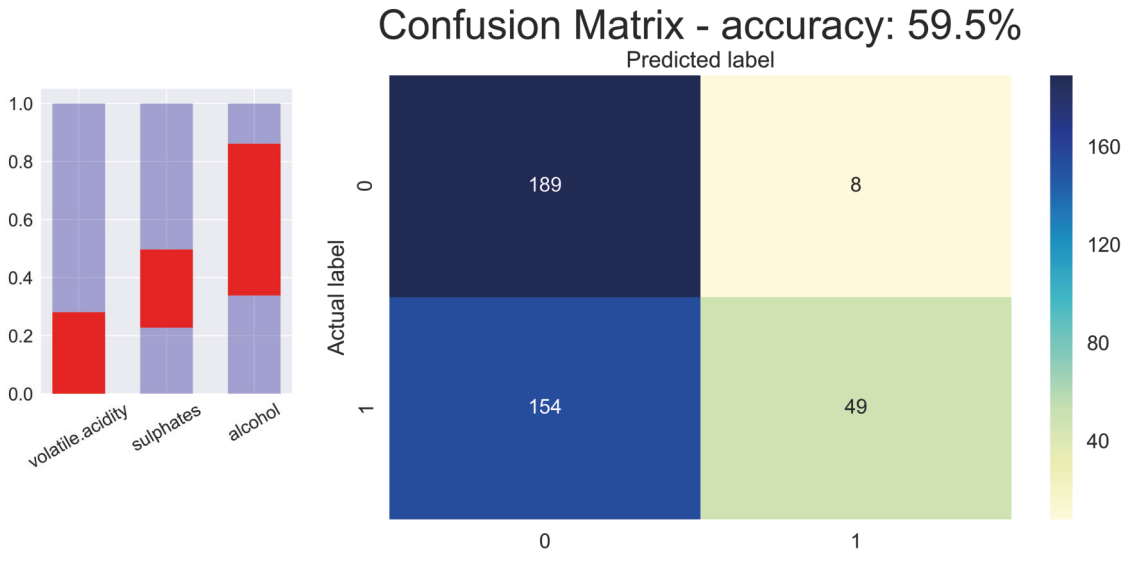


Figure 8.19: Interval 2 (decision tree - gini): 133 wines, 133 good and 0 bad

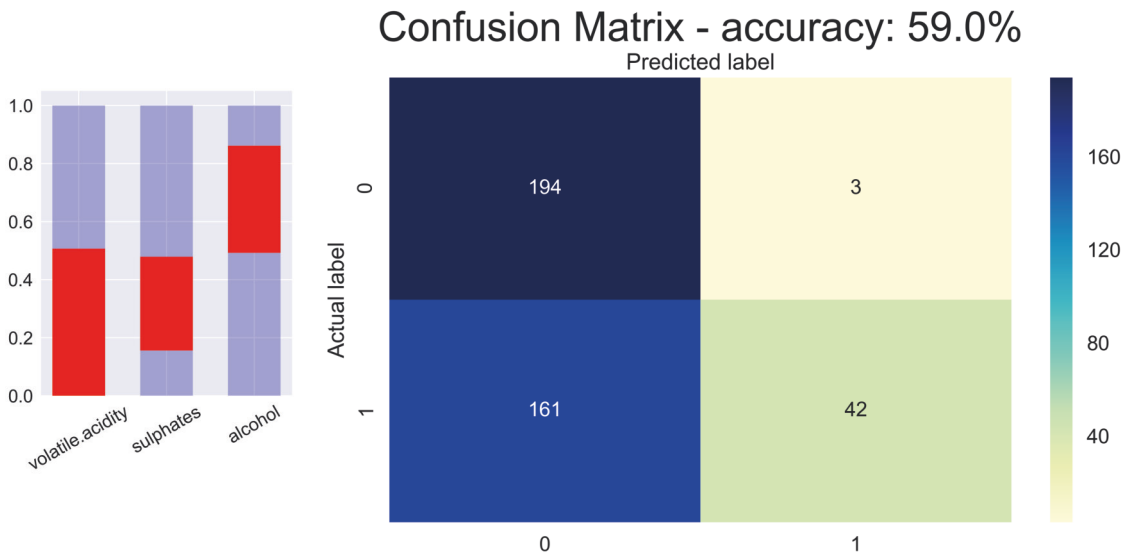


Figure 8.20: Interval 3 (decision tree - gini): 133 wines, 133 good and 0 bad

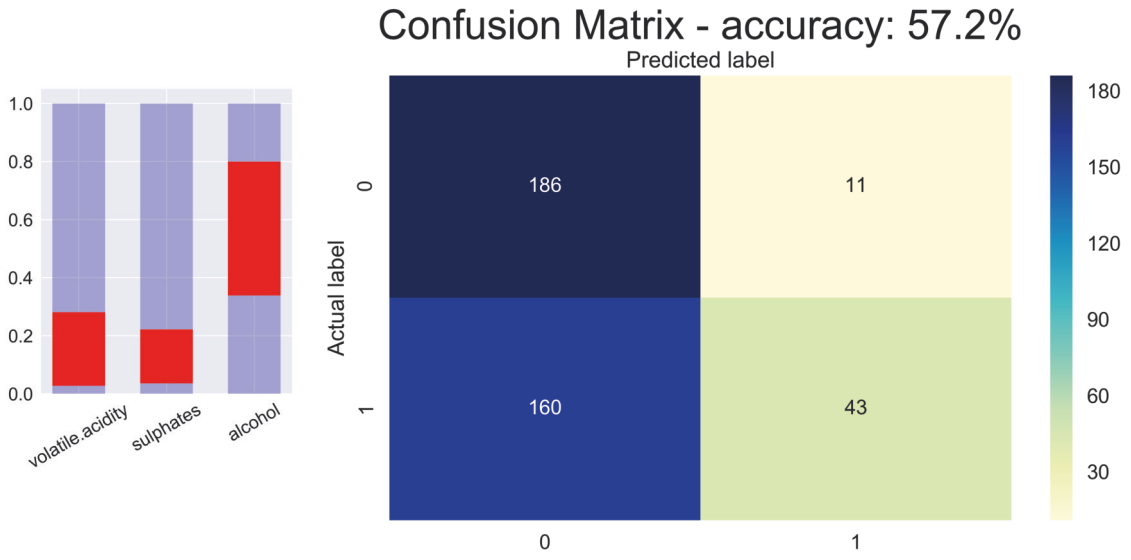


Figure 8.21: Interval 4 (k-medoids - euclidean): 123 wines, 123 good and 0 bad

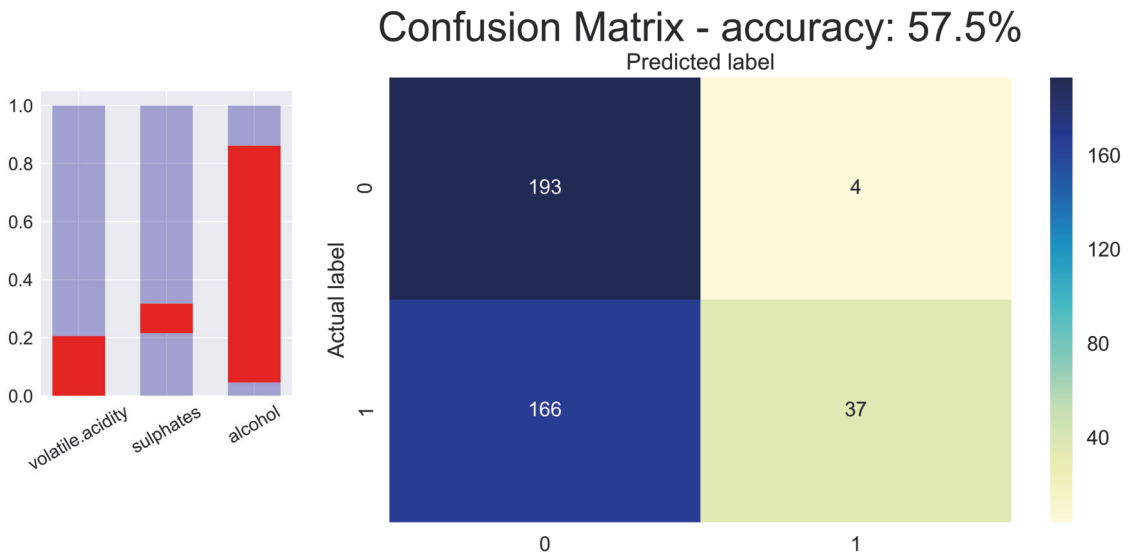


Figure 8.22: Interval 5 (k-medoids - mahalanobis): 120 wines, 120 good and 0 bad

Combining these 5 intervals to predict the test set we receive the following result.

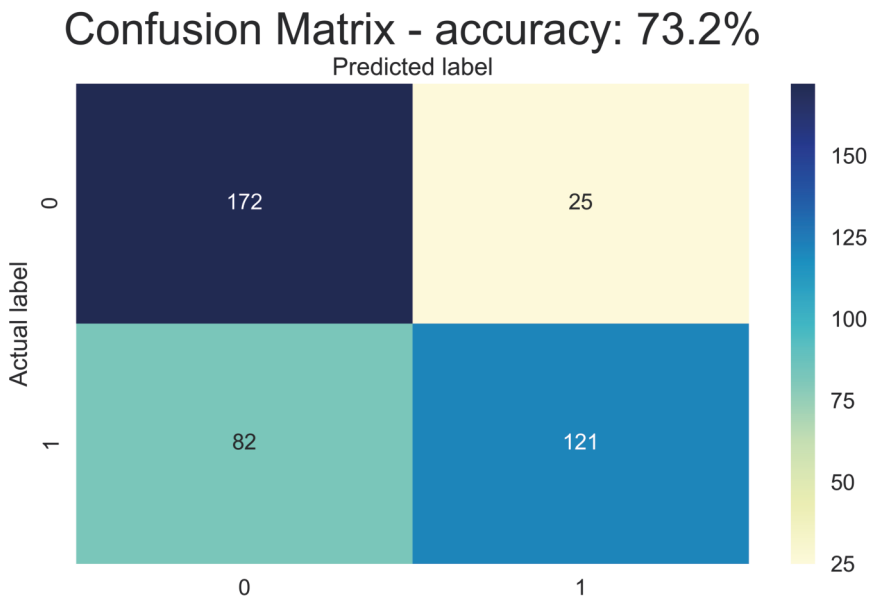


Figure 8.23: Predicting the test set with the 5 intervals

Adding other attributes does not provide any predictive benefit. However, we can conclude that a reduction on the attribute set does not decrease our forecast accuracy on the contrary, it leads to an improvement. For comparison: A random forest trained on the trainings set, which is used in this section, leads to the confusion matrix below on the test set.

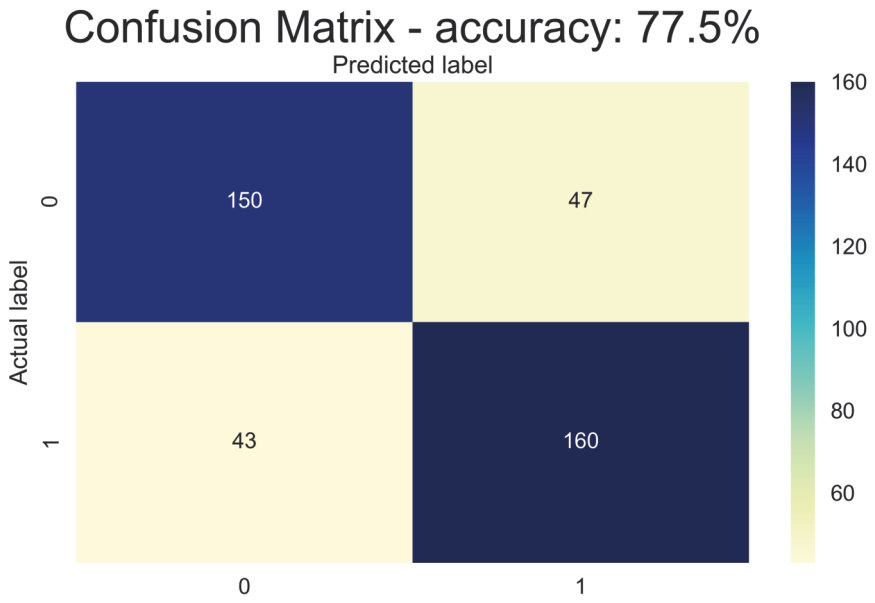


Figure 8.24: Prediction on the red wine test set using a random forest consisting of 100 trees

As in the last chapter, the predictive quality of our model decreases somewhat compared to the random forest, but the high interpretability also makes the here presented algorithm useful.

8.3 Conclusion and Critical Discussion

The code under Section C offers the possibility to use even more clustering algorithms than shown in this section. We selected the here presented collection, because they delivered the best results on the test set in test runs. Further investigations are needed to find the best collections of clustering algorithms for different use cases (e.g. maximize accuracy) and data sets. The specifications of our algorithm also need to be checked. We tried different specifications via grid search e.g. the range of the randomly chosen number of clusters, number of wines in the intervals, number of intervals, which were used for the model. As in the last chapter the approach presented on the previous pages is just an example of building a model from our framework. The greatest knowledge of this chapter is that we can find important patterns in a pattern structure through clustering algorithms because they are an important field of research (for example see [VM02], [SQT02], [CH74], [KS96], [ESBB98]). Hopefully, further investigations show that it is possible to create even stronger models with our framework and also find specifications, that can be applied to a large number of data sets. As shown here, the idea

of building interval patterns from clustering algorithms is capable of giving good predictions and the good interpretability makes it even more powerful.

9 Conclusion

This thesis is divided into a theoretical part, aimed at developing statements around the newly introduced concept of pattern morphisms, and a practical part, where we present use cases of pattern structures.
A first insight of our work clarifies the facts on projections of pattern structures. We discovered that a projection of a pattern structure does not always lead again to a pattern structure. A solution to this problem, and one of the most important points of this thesis, is the introduction of pattern morphisms in Chapter 4. Pattern morphisms make it possible to describe relationships between pattern structures, and thus enable a deeper understanding of pattern structures in general. They also provide the means to describe projections of pattern structures that lead to pattern structures again. In Chapter 5 and 6, we looked at the impact of morphisms between pattern structures on concept lattices and on their representations and thus clarified the theoretical background of existing research in this field. The theory on pattern morphisms stands on a solid foundation now, which we consider another major result of our work. However, the connection to other research still needs to be investigated further. As mentioned in Chapter 2, there is a connection between interval pattern structures and fuzzy formal contexts, as, for example, presented in [PK12], where interval pattern concepts are shown to be related to certain formal concepts. It might be worth looking at this connection from a pattern morphism perspective. We also believe that future works on chains of projections (e.g. see [BKN17b]) could benefit from approaching the topic from a pattern morphism angle.
The application part reveals that random forests can be described through pattern structures, which constitutes another central achievement of our work. In order to demonstrate the practical relevance of our findings, we included a use case where this finding is used to build an algorithm that solves a real world classification problem of red wines. The prediction accuracy of the random forest is better, but the high interpretability makes our algorithm valuable. Another approach to the red wine classification problem is presented in Chapter 8, where, starting from an elementary pattern structure, we built a classification model that yielded good results. We hope that further investigations on other data sets confirm that the two practical approaches are useful and that the powerful theory on pattern morphisms leads to interesting research results in the future.

Index

Bibliography

[AV06] D. Arthur and S. Vassilvitskii. *k-means++: The Advantages of Careful Seeding*. Technical Report 2006-13, Stanford InfoLab, 2006.

[BBK19] Aimene Belfodil, Adnene Belfodil, and M. Kaytoue. *Mining Formal Concepts using Implications between Items*. In *Proceedings of the 15th International Conference on Formal Concept Analysis*, pages 173–190. Springer, 2019.

[BBOV09] R. Belohlavek, B. De Baets, J. Outrata, and V. Vychodil. *Inducing decision trees via concept lattices*. *International journal of general systems*, 38(4):pages 455–467, 2009.

[BD16] M. Belgiu and L. Drăguţ. *Random forest in remote sensing: A review of applications and future directions*. *ISPRS Journal of Photogrammetry and Remote Sensing*, volume 114:pages 24–31, 2016.

[Bel11] R. Belohlavek. *What is a fuzzy concept lattice? .* In *International Workshop on Rough Sets, Fuzzy Sets, Data Mining, and Granular-Soft Computing*, pages 19–26. Springer, 2011.

[Bel12] R. Belohlavek. *Fuzzy relational systems: foundations and principles*, volume 20. Springer Science & Business Media, 2012.

[Bel19a] Adnene Belfodil. *Exceptional Model Mining for Behavioral Data Analysis*. PhD thesis, 2019.

[Bel19b] Aimene Belfodil. *An Order Theoretic Point-of-view on Subgroup Discovery*. PhD thesis, 2019.

[BFSO84] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[BKK18] Aimene Belfodil, S. Kuznetsov, and M. Kaytoue. *Pattern setups and their completions*. In *Proceedings of the 14th International Conference on Concept Lattices and Their Applications*, pages 243–253, 2018.

[BKK20] Aimene Belfodil, S. Kuznetsov, and M. Kaytoue. *On Pattern Setups and Pattern Multistructures*. *International Journal of General Systems*, 49:pages 785–818, 2020.

[BKN15a] A. Buzmakov, S. O. Kuznetsov, and A. Napoli. *Fast generation of best interval patterns for nonmonotonic constraints*. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 157–172. Springer, 2015.

[BKN15b] A. Buzmakov, S. O. Kuznetsov, and A. Napoli. *Revisiting Pattern Structure Projections*. In *Proceedings of the 13th International Conference on Formal Concept Analysis*, volume 9113, pages 200–215, 2015.

[BKN17a] A. Buzmakov, S. O. Kuznetsov, and A. Napoli. *Efficient mining of subsample-stable graph patterns*. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 757–762. IEEE, 2017.

[BKN17b] A. Buzmakov, S. O. Kuznetsov, and A. Napoli. *Mining best closed itemsets for projection-antimonotonic constraints in polynomial time*. *arXiv preprint arXiv:1703.09513*, 2017.

[Bre01] L. Breiman. *Random forests*. *Machine learning*, volume 45(1):pages 5–32, 2001.

[CCA+09] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. *Modeling wine preferences by data mining from physicochemical properties*. In *Decision Support Systems*, volume 47, pages 547–553. Elsevier, 2009.

[CCOGP+16] I. P. Cabrera, P. Cordero-Ortega, F. García-Pardo, M. Ojeda-Aciego, and B. De Baets. *On the existence of right adjoints for surjective mappings between fuzzy structures*. 2016.

[CH74] T. Caliński and J. Harabasz. *A dendrite method for cluster analysis*. *Communications in Statistics-theory and Methods*, volume 3(1):pages 1–27, 1974.

[DUA06] R. Díaz-Uriarte and S. A. De Andres. *Gene selection and classification of microarray data using random forest*. *BMC bioinformatics*, volume 7(1):pages 3–16, 2006.

[Ern04] M. Erné. *Adjunctions and Galois Connections: Origins, History and Development*. In *Galois Connections and Applications*, Mathematics and Its Applications. Springer Netherlands, 2004.

[ESBB98] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. *Cluster analysis and display of genome-wide expression patterns*. *Proceedings of the National Academy of Sciences*, volume 95(25):pages 14863–14868, 1998.

[GBV+08] S. Guillas, K. Bertet, M. Visani, J. Ogier, and N. Girard. *Some links between decision tree and dichotomic lattice*. In *Proceedings of the Sixth International Conference on Concept Lattices and Their Applications*, pages 193–205. Citeseer, 2008.

[GK01] B. Ganter and S. Kuznetsov. *Pattern Structures and Their Projections*. In *Proceedings of the 9th International Conference on Conceptual Structures.*

Lecture Notes in Artificial Intelligence (Springer), volume 2120, pages 129–142, 2001.

[GW13] B. Ganter and R. Wille. *Formal Context Analysis: Mathematical Foundations*. Springer Berlin Heidelberg, 2013.

[JFG94] A. B. Juandeaburre and R. Fuentes-González. *The study of the L-fuzzy concept lattice*. *Mathware & soft computing. 1994 Vol. 1 Núm. 3*, pages 209–218, 1994.

[KKND11] M. Kaytoue, S. O. Kuznetsov, A. Napoli, and S. Duplessis. *Mining gene expression data with pattern structures in formal concept analysis*. In *Information Sciences (Elsevier)*, volume 181, pages 1989–2001, 2011.

[KS96] D. J. Ketchen and C. L. Shook. *The application of cluster analysis in strategic management research: an analysis and critique*. *Strategic management journal*, volume 17(6):pages 441–458, 1996.

[KS11] B. Kaiser and S. E. Schmidt. *Some remarks on the relation between annotated ordered sets and pattern structures*. In *Proceedings of the 5th international Conference on Pattern Recognition and Machine Intelligence. Lecture Notes in Computer Science (Springer)*, volume 6744, pages 43–48, 2011.

[Kuz04] S. O. Kuznetsov. *Machine learning and formal concept analysis*. In *Proceedings of the 2nd International Conference on Formal Concept Analysis*, pages 287–312. Springer, 2004.

[Kuz09] S. O. Kuznetsov. *Pattern structures for analyzing complex data*. In *Proceedings of the 12th International Conference on Rough Sets, Fuzzy Sets, Data Mining and Granular Computing. Lecture Notes in Artificial Intelligence (Springer)*, volume 5908, pages 33–44, 2009.

[Kuz13] S. O. Kuznetsov. *Scalable Knowledge Discovery in Complex Data with Pattern Structures*. In *Proceedings of the 5th International Conference on Pattern Recognition and Machine Intelligence. Lecture Notes in Computer Science (Springer)*, volume 8251, pages 30–41, 2013.

[LS15a] L. Lumpe and S. E. Schmidt. *A Note on Pattern Structures and Their Projections*. In *Proceedings of the 13th International Conference on Formal Concept Analysis*, volume 9113, pages 145–150, 2015.

[LS15b] L. Lumpe and S. E. Schmidt. *Pattern Structures and Their Morphisms*. In *Proceedings of the 12th International Conference on Concept Lattices and Their Applications*, pages 171–179, 2015.

[LS16] L. Lumpe and S. E. Schmidt. *Morphisms Between Pattern Structures and Their Impact on Concept Lattices*. In *Proceedings of the 22th European Conference on Artificial Intelligence (Workshop FCA4AI)*, pages 25–34, 2016.

[LS17] L. Lumpe and S. E. Schmidt. *Viewing Morphisms Between Pattern Structures via Their Concept Lattices and via Their Representations*. In *Proceedings of the 23th International Symposium on Methodologies for Intelligent Systems*, pages 597–608, 2017.

[LS20a] L. Lumpe and S. E. Schmidt. *A Link Between Pattern Structures and Random Forests*. In *Proceedings of the 15th International Conference on Concept Lattices and Their Applications*, pages 131–143, 2020.

[LS20b] L. Lumpe and S. E. Schmidt. *Patterns via Clustering as a Data Mining Tool*. In *Proceedings of the 24th European Conference on Artificial Intelligence (Workshop FCA4AI)*, pages 81–89, 2020.

[Mad12] T. S. Madhulatha. *An overview on clustering methods*. *arXiv preprint arXiv:1205.1117*, 2012.

[Mah05] P. Mahesh. *Random forest classifier for remote sensing classification*. *International journal of remote sensing*, volume 26(1):pages 217–222, 2005.

[Pig10] A. Pigors. *Allgemeine Residuiertheit und Hüllenstrukturen*. PhD thesis, Gottfried Wilhelm Leibniz Universität Hannover, 2010.

[PJ09] H. S. Park and C. H. Jun. *A simple and fast algorithm for K-medoids clustering*. In *Expert Systems with Applications (Elsevier)*, volume 36 (2), pages 3336–3341, 2009.

[PK12] V. V. Pankratieva and S. O. Kuznetsov. *Relations between proto-fuzzy concepts, crisply generated fuzzy concepts, and interval pattern structures*. *Fundamenta Informaticae*, 115(4):pages 265–277, 2012.

[RGGR+12] V. F. Rodriguez-Galiano, B. Ghimire, J. Rogan, M. Chica-Olmo, and J. P. Rigol-Sanchez. *An assessment of the effectiveness of a random forest classifier for land-cover classification*. *ISPRS Journal of Photogrammetry and Remote Sensing*, volume 67:pages 93–104, 2012.

[RM05] L. Rokach and O. Maimon. *Top-down induction of decision trees classifiers-a survey*. volume 35, pages 476–487. IEEE, 2005.

[SL91] S. R. Safavian and D. Landgrebe. *A survey of decision tree classifier methodology*. volume 21, pages 660–674. IEEE, 1991.

[SLT+03] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston. *Random forest: a classification and regression tool for compound classification and QSAR modeling*. *Journal of chemical information and computer sciences*, volume 43(6):pages 1947–1958, 2003.

[SQT02] A. Sturn, J. Quackenbush, and Z. Trajanoski. *Genesis: cluster analysis of microarray data*. *Bioinformatics*, volume 18(1):pages 207–208, 2002.

[SS14] H. Soldano and G. Santini. *Graph abstraction for closed pattern mining in attributed networks*. In *ECAI*, pages 849–854, 2014.

[VM02] J. K. Vermunt and J. Magidson. *Latent class cluster analysis*. *Applied latent class analysis*, volume 11(60):pages 89–106, 2002.

Appendices

A: Get Random Forest

This code was used to produce Figure 7.14, Figure 7.15, Figure 7.16, Figure 7.17, Figure 7.18, Figure 7.19, Figure 7.20.

```
import pandas as pd
import numpy as np
import seaborn as sns
import random
import matplotlib
import sympy
import sklearn.tree
import csv


from numpy import *
from matplotlib import pyplot as plt
from sklearn import preprocessing  #skaliert features zwischen -1 und 1 fÃ¼r
                                        schnellere Berechnung
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics.pairwise import pairwise_distances
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from datetime import date
from itertools import *

from sklearn.cluster import KMeans
from sklearn.cluster import AffinityPropagation
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.cluster import SpectralClustering
from sklearn.cluster import DBSCAN
from sklearn.cluster import Birch
from sklearn.cluster import estimate_bandwidth
from sklearn.cluster import MeanShift
from sklearn.mixture import GaussianMixture
from sklearn.mixture import BayesianGaussianMixture

from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler


from sklearn.tree import export_graphviz
```

```
import plotly
import plotly.graph_objs as go


import operator

from sklearn import tree
import graphviz




plotly.offline.init_notebook_mode()
plt.style.use('seaborn')
random.seed(42)




from random import seed
from random import random


import pickle


def show_tree(df,n):
    feature_names = df.drop(["quality"],axis=1).columns
    class_names = ["bad", "good"]
    treeIndex = n

    dot_data = tree.export_graphviz(clf.estimators_[treeIndex], out_file=None,
                                    feature_names=feature_names,
                                    class_names=class_names,
                                    filled=True, rounded=True,
                                    special_characters=True)

    return graphviz.Source(dot_data)

def show_and_save_cfm(y_test, y_tree):

    cnf_matrix = metrics.confusion_matrix(y_test, y_tree)

    # name  of classes
    class_names=[0,1]

    fig, ax = plt.subplots()
    tick_marks = np.arange(len(class_names))
```

```
    plt.xticks(tick_marks, class_names)
    plt.yticks(tick_marks, class_names)

    acc_tree = metrics.accuracy_score(y_test, y_tree)

    acc_tree = acc_tree * 100
    # create heatmap
    sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu", fmt='g')
    ax.xaxis.set_label_position("top")
    plt.tight_layout()
    plt.title('Confusion matrix - accuracy: '+str(acc_tree.round(1)) + '%', y=1.1,
                                                fontsize =20)
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')
    fig.tight_layout()
    fig.savefig('confusion total ' + 'step 1'  + '.png', dpi=600)
    plt.show()


# data import
df = pd.read_excel (r'C:\Users\larsl\Desktop\Mathe\Python\red-wine-dataset\
                                        wineQualityReds01.xlsx')
# Drop Null/NA Values from DataFrame
df.dropna(inplace = True)


X = df
y = np.array(df['quality'])
X = X.drop(['quality'],axis = 1)
labels = X.columns
X = np.array(X)

# Seperate good and bad
good = df[df.quality==1]
X_good = good.drop(['quality'],axis = 1)
X_good = np.array(X_good)

bad = df[df.quality==0]
X_bad = bad.drop(['quality'],axis = 1)
X_bad = np.array(X_bad)

# Number of attributes (11)
numa=len(labels)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,
                                        random_state = 4)

#X_train_good
X_train_good = X_train[y_train == 1]
```

```
#X_train_bad
X_train_bad = X_train[y_train == 0]

X_test = np.array(X_test)
X_train_good = np.array(X_train_good)
X_train_bad = np.array(X_train_bad)

num_est = 1

clf = RandomForestClassifier(n_estimators = num_est, min_samples_leaf = 10,
                                          max_depth = 3)


clf.fit(X_train, y_train)


y_forest = clf.predict(X_test)




#Confusion Matrix
show_and_save_cfm(y_test, y_forest)


graph=show_tree(df,0)
graph.render('decision_tree', view=True)
graph
```

B: Get Intervals from Random Forest

This code was used to produce Figure 7.21, Figure 7.22, Figure 7.23, Figure 7.24, Figure 7.25, Figure 7.26, Figure 7.27, Figure 7.28, Figure 7.29, Figure 7.30, Figure 7.31, Figure 7.32, Figure 7.33, Figure .1, Figure .2, Figure .3, Figure .4, Figure .5, Figure .6, Figure .7, Figure .8 and Figure .9.

```
# Import Modules

import pandas as pd
import numpy as np
import seaborn as sns
import random
import matplotlib
import sympy
import sklearn.tree
import csv


from numpy import *
from matplotlib import pyplot as plt
from sklearn import preprocessing
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics.pairwise import pairwise_distances
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from datetime import date
from itertools import *

from sklearn.cluster import KMeans
from sklearn.cluster import AffinityPropagation
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.cluster import SpectralClustering
from sklearn.cluster import DBSCAN
from sklearn.cluster import Birch
from sklearn.cluster import estimate_bandwidth
from sklearn.cluster import MeanShift
from sklearn.mixture import GaussianMixture
from sklearn.mixture import BayesianGaussianMixture

from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
```

```
from sklearn.preprocessing import MinMaxScaler

from sklearn.tree import export_graphviz


import plotly
import plotly.graph_objs as go

import operator


from sklearn import tree
import graphviz




plotly.offline.init_notebook_mode()
plt.style.use('seaborn')
random.seed(42)




from random import seed
from random import random


import pickle


# Functions

def show_Balken(MinList,MaxList):

    n = len(MinList)
    m = len(MinList[0])

    min_examples = np.ones(m)
    fig, ax = plt.subplots(figsize=(18,5))
    plt.bar(labels, min_examples, width = 0.5, color = 'b', alpha=.3)

    for d in range(n):

            for j in range(m):

                oben  = MaxList[d][j]
                unten = MinList[d][j]

                this_alpha=1
```

```
                plt.bar(labels[j], oben - unten, width = 0.5, color = 'r',alpha =
                                                        this_alpha ,bottom =
                                                        unten)

def show_tree(df,n):
    feature_names = df.drop(["quality"],axis=1).columns
    class_names = ["bad", "good"]
    treeIndex = n

    dot_data = tree.export_graphviz(clf.estimators_[treeIndex], out_file=None,
                                    feature_names=feature_names,
                                    class_names=class_names,
                                    filled=True, rounded=True,
                                    special_characters=True)

    return graphviz.Source(dot_data)

def get_pattern_matrices (M,label):

    if type(label) is dict:
        m = len(M[0])
        P_max = np.zeros([1,m],dtype=int)
        P_min = np.zeros([1,m],dtype=int)

        for lab in label:
            P_max_col = [0]
            P_max_col[0] = max(M[label[lab]][:,0])
            for col in range(1,m):
                P_max_col.append(max(M[label[lab]][:,col]))

            P_max = np.vstack([P_max,P_max_col])

        P_max = np.delete(P_max, (0), axis=0)


        for lab in label:
            P_min_col = [0]
            P_min_col[0] = min(M[label[lab]][:,0])
            for col in range(1,m):
                P_min_col.append(min(M[label[lab]][:,col]))

            P_min = np.vstack([P_min,P_min_col])

        P_min = np.delete(P_min, (0), axis=0)

    else:

        n = len(M[0])
        #print(n)
        max_label = max(label)
```

```
min_label = min(label)
label_output = [0]

P_max = np.zeros([1,n],dtype=int)
P_min = np.zeros([1,n],dtype=int)


P_max_col = [0]
P_max_col[0] = max(M[label == min_label][:,0])


P_min_col = [0]
P_min_col[0] = min(M[label == min_label][:,0])


for l in range(max_label + 1):
    M_l = np.array(M[label==l])

    if M_l.size == 0:
        continue

    P_max_col = [0]
    P_max_col[0] = max(M_l[:,0])
    for col in range(1,n):
        P_max_col.append(max(M_l[:,col]))



    P_max = np.vstack([P_max,P_max_col])
    label_output.append(l)

P_max = np.delete(P_max, (0), axis=0)
label_output = np.delete(label_output, (0), axis=0)

for l in range(max_label + 1):
    M_l = np.array(M[label==l])

    if M_l.size == 0:
        continue

    P_min_col = [0]
    P_min_col[0] = min(M_l[:,0])
    for col in range(1,n):
        P_min_col.append(min(M_l[:,col]))



    P_min = np.vstack([P_min,P_min_col])

P_min = np.delete(P_min, (0), axis=0)
```

```
    return P_max, P_min, label_output

def countcustomerinpattern(customer,P_min,P_max):
    pattern = len(P_min)
    attributes = len(P_min[0])
    customer_count = len(customer)
    counter = np.zeros(pattern)



    for p1 in range(pattern):
        for c in range(customer_count):
                for a in range(attributes):
                    if customer[c][a] < P_min[p1][a] or customer[c][a] > P_max[p1]
                                                                        [a]:
                        break

                    if a == (attributes-1):
                        counter[p1] += 1


    return counter

def countcustomerspattern(customer,P_min,P_max):

    attributes = len(P_min[0])

    if P_min.size == 0:

        print('irgendwie hier gelandet')
        customer_count = len(customer)
        counter = np.zeros(customer_count)

        return counter


    else:

        try:
            pattern = len(P_min)
        except:
            pattern = P_min.shape[0]


        customer_count = len(customer)
        counter = np.zeros(customer_count)

        for c in range(customer_count):
            #print('c')
```

```
            #print(c)

            if pattern == 0:

                for p1 in range(pattern + 1):

                    #print('p1a')
                    #print(p1)
                    for a in range(attributes):
                        #print('aa')
                        #print(a)
                        if customer[c,a] < P_min[a] or customer[c,a] > P_max[a]:
                            break
                        if a == (attributes-1):
                            counter[c] = counter[c] + 1


            else:

                for p1 in range(pattern):

                    #print('p2')
                    #print(p1)

                    for a in range(attributes):

                        if customer[c,a] < P_min[p1,a] or customer[c,a] > P_max[p1
                                                                ,a]:
                            break

                        if a == (attributes-1):

                            counter[c] = counter[c] + 1


        return counter

def show_and_save_cfm(y_test, y_tree):

    cnf_matrix = metrics.confusion_matrix(y_test, y_tree)

    # name  of classes
    class_names=[0,1]

    fig, ax = plt.subplots()
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names)
    plt.yticks(tick_marks, class_names)

    acc_tree = metrics.accuracy_score(y_test, y_tree)
```

```
    acc_tree = acc_tree * 100
    # create heatmap
    sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu", fmt='g')
    ax.xaxis.set_label_position("top")
    plt.tight_layout()
    plt.title('Confusion Matrix - accuracy: '+str(acc_tree.round(1)) + '%', y=1.1,
                                                fontsize =20)
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')
    fig.tight_layout()
    fig.savefig('confusion total' + '.png', dpi=600)
    plt.show()

# Script

## Import Data

# Import data
df = pd.read_excel (r'C:\Users\larsl\Desktop\Mathe\Python\red-wine-dataset\
                                            wineQualityReds01.xlsx')
# Drop Null/NA Values from DataFrame
df.dropna(inplace = True)

X = df
y = np.array(df['quality'])
X = X.drop(['quality'],axis = 1)
labels = X.columns
X = np.array(X)

# Seperate good and bad
good = df[df.quality==1]
X_good = good.drop(['quality'],axis = 1)
X_good = np.array(X_good)

bad = df[df.quality==0]
X_bad = bad.drop(['quality'],axis = 1)
X_bad = np.array(X_bad)

# Number of attributes (11)
numa=len(labels)

## MinMaxScaler

mms = MinMaxScaler()
mms.fit(X)

#X_bad_scale = mms.transform(X_bad)

X_scale = mms.transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.25,
                                                    random_state = 4)

#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,
                                                    random_state = 4)

#X_train_good
X_train_good = X_train[y_train == 1]
#X_train_bad
X_train_bad = X_train[y_train == 0]

X_test = np.array(X_test)
X_train_good = np.array(X_train_good)
X_train_bad = np.array(X_train_bad)

## Random Forest

num_est = 10

clf = RandomForestClassifier(n_estimators = num_est, min_samples_leaf = 10,
                                                    max_depth = 5, random_state = 20)

clf.fit(X_train, y_train)

y_forest = clf.predict(X_test)

show_and_save_cfm(y_test, y_forest)

graph=show_tree(df,0)
graph.render('decision_tree', view=True)
graph

## Dictionaries

P_min_dict = {}
P_max_dict = {}
label_output_dict = {}
ratio_dict = {}
cluster_dict = {}
counter_g_dict = {}
counter_b_dict = {}

for k in range(num_est):

    cluster_total_tree = clf.estimators_[k].tree_.apply(np.asfortranarray(X_train.
                                                        astype(sklearn.tree._tree.DTYPE)))

    P_max,P_min,label_output = get_pattern_matrices(X_train,cluster_total_tree)
```

```
    counter_g = countcustomerinpattern(X_train_good,P_min,P_max)
    counter_b = countcustomerinpattern(X_train_bad,P_min,P_max)

    ratio = np.divide(counter_g,(counter_g + counter_b), out = np.zeros_like(
                                            counter_g), where =counter_g!=0)


    P_min_dict[k]        = P_min
    P_max_dict[k]        = P_max
    label_output_dict[k] = label_output
    counter_g_dict[k]    = counter_g
    counter_b_dict[k]    = counter_b
    ratio_dict[k]        = ratio
    cluster_dict[k]      = cluster_total_tree

## Calculate Aggregates

P_min_agg = np.ones([len(X_train),len(P_min_dict[0][0])])
P_max_agg = np.zeros([len(X_train),len(P_min_dict[0][0])])

P_min_agg_th = np.ones([len(X_train),len(P_min_dict[0][0])])
P_max_agg_th = np.zeros([len(X_train),len(P_min_dict[0][0])])

treshold_ratio = 0.5


for i in range(len(X_train)):
    for j in range(len(P_min_dict[0][0])):

        for d in P_min_dict:

                P_min_agg[i,j] = min(P_min_agg[i,j], P_min_dict[d][
                                                    label_output_dict[d] ==
                                                    cluster_dict[d][i],j])

                if ratio_dict[d][label_output_dict[d] == cluster_dict[d][i]] >
                                                    treshold_ratio:

                    P_min_agg_th[i,j] = min(P_min_agg[i,j], P_min_dict[d][
                                                        label_output_dict[d]
                                                        == cluster_dict[d][i]
                                                        ,j])


for i in range(len(X_train)):
    for j in range(len(P_max_dict[0][0])):

        for d in P_max_dict:
```

```
                P_max_agg[i,j] = max(P_max_agg[i,j], P_max_dict[d][
                                                        label_output_dict[d] ==
                                                        cluster_dict[d][i],j])

                if ratio_dict[d][label_output_dict[d] == cluster_dict[d][i]] >
                                                        treshold_ratio:

                    P_max_agg_th[i,j] = max(P_max_agg[i,j], P_max_dict[d][
                                                            label_output_dict[d]
                                                            == cluster_dict[d][i]
                                                            ,j])

counter_g_agg = countcustomerinpattern(X_train_good,P_min_agg,P_max_agg)
counter_b_agg = countcustomerinpattern(X_train_bad,P_min_agg,P_max_agg)

counter_g_agg_th = countcustomerinpattern(X_train_good,P_min_agg_th,P_max_agg_th)
counter_b_agg_th = countcustomerinpattern(X_train_bad,P_min_agg_th,P_max_agg_th)

ratio_agg = np.divide(counter_g_agg,(counter_g_agg + counter_b_agg), out = np.
                                        zeros_like(counter_g_agg), where =
                                        counter_b_agg!=0)
ratio_agg_th = np.divide(counter_g_agg_th,(counter_g_agg_th + counter_b_agg_th),
                                        out = np.zeros_like(counter_g_agg_th),
                                        where =counter_b_agg_th!=0)

print(ratio_agg_th)

## Sort By Ratio

ratio_agg = ratio_agg.reshape(-1,1)

counter_g_sorted = [x for _,x in sorted(zip(ratio_agg,counter_g_agg),reverse =
                                        True)]
counter_b_sorted = [x for _,x in sorted(zip(ratio_agg,counter_b_agg),reverse =
                                        True)]

ratio_agg_th_sorted = [x for _,x in sorted(zip(ratio_agg,ratio_agg_th),reverse =
                                        True)]

ratio_sorted_unique = sorted(unique(ratio_agg), reverse = True)


#for i in range(10):
#    print(str(i+1) + '. ratio: ' + str(ratio_sorted[i]) + ' good: ' + str(
                                        counter_g_sorted[i])+ ' bad: ' + str(
                                        counter_b_sorted[i]) )

#number of pattern
counter_p = 5
```

```
best_ratio = ratio_sorted_unique[0:counter_p]

#to select the worst ratio:
#best_ratio = ratio_sorted_unique[-1:]

print(worst_ratio)


for i in range(counter_p):

    print('')
    print(str(i+1) + ". best ratio is: " + str(best_ratio[i]))

    ratio_pos = ratio_agg == best_ratio[i]

    ratio_with_treshold = ratio_agg_th[ratio_pos[:,0]]
    ratio_with_treshold = ratio_with_treshold[0]

    print('ratio with treshold: ' + str(ratio_with_treshold))


## Show Pattern

min_examples = X_train.max(axis = 0)
min_examples = np.ones(numa)

Forest_HQ_Min = []
Forest_HQ_Max = []

Meta_Forest_Min = np.zeros([1,numa])
Meta_Forest_Min = Meta_Forest_Min[0]
Meta_Forest_Max = np.zeros([1,numa])
Meta_Forest_Max = Meta_Forest_Max[0]
print(Meta_Forest_Min[0])



for i in range(counter_p):

    print('best ratio: ' + str(best_ratio[i]))

    #Position des tollen Weins in der Liste
    pos_wine  = np.where(ratio_agg == best_ratio[i])
    pos_wine  = pos_wine[0][0]

    Forest_HQ_Min = P_min_agg[pos_wine,:]
    Forest_HQ_Max = P_max_agg[pos_wine,:]


    fig, ax = plt.subplots(figsize=(18,5))
```

```
    plt.bar(labels, min_examples, width = 0.5, color = 'b', alpha=.3)

    number_examples = counter_g_agg[pos_wine] + counter_b_agg[pos_wine]
    print('recognized: ' + str(number_examples))

    number_good_examples = counter_g_agg[pos_wine]
    print('good: ' + str(number_good_examples))

    ratio_examples = number_good_examples/number_examples
    print('ratio: ' + str(ratio_examples))


    for j in range(numa):

        #this_alpha = min ( 1, max(0,((anz)/anz_gesamt))+0.2 )
        this_alpha = 1
        plt.bar(labels[j], P_max_agg[pos_wine,j] - P_min_agg[pos_wine,j], width =
                                                0.5, color = 'r',alpha =
                                                this_alpha ,bottom = P_min_agg[
                                                pos_wine,j])

    plt.xticks(labels, rotation = 30, fontsize = 16)
    plt.yticks(fontsize = 16)
    plt.savefig(str(i) + '. best pattern' + str(best_ratio[i])+ '_' +str(
                                            number_examples)+ '_' + str(
                                            number_good_examples) + '.jpg', dpi =
                                             600, bbox_inches='tight')
    plt.show()

    Meta_Forest_Min = np.vstack((Meta_Forest_Min, Forest_HQ_Min))
    Meta_Forest_Max = np.vstack((Meta_Forest_Max, Forest_HQ_Max))


Meta_Forest_Min = np.delete(Meta_Forest_Min, (0), axis=0)
Meta_Forest_Max = np.delete(Meta_Forest_Max, (0), axis=0)


y_pred_count_meta = countcustomerspattern(X_test, Meta_Forest_Min, Meta_Forest_Max
                                        )
y_pred_meta = y_pred_count_meta
y_pred_meta[y_pred_meta < 1]=0
y_pred_meta[y_pred_meta > 0]=1

show_and_save_cfm(y_test, y_pred_meta)
```

B.1: Examples of Intervals

Here we present intervals of the best scored wine of different random forests. The data we used to build these random forests is presented in Section 7.1 and the proceeding is described in Section 7.4. We build the random forest on the trainingset and predicted the classes of the testset. In the first example we used a random forest with 2 decision trees and got the following result:

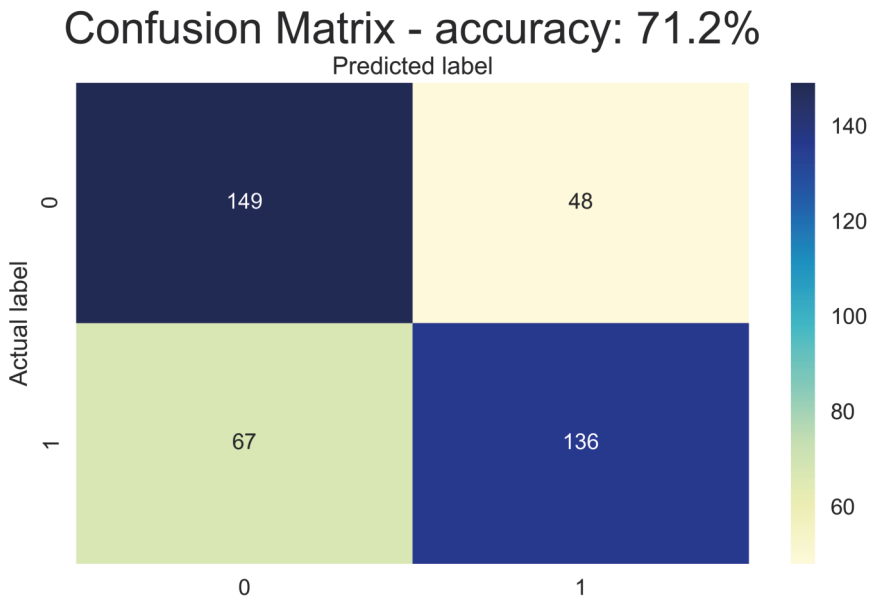


Figure .1:: random forest build from 2 decision trees

The best scored wine of the random forest lead us to the following interval:

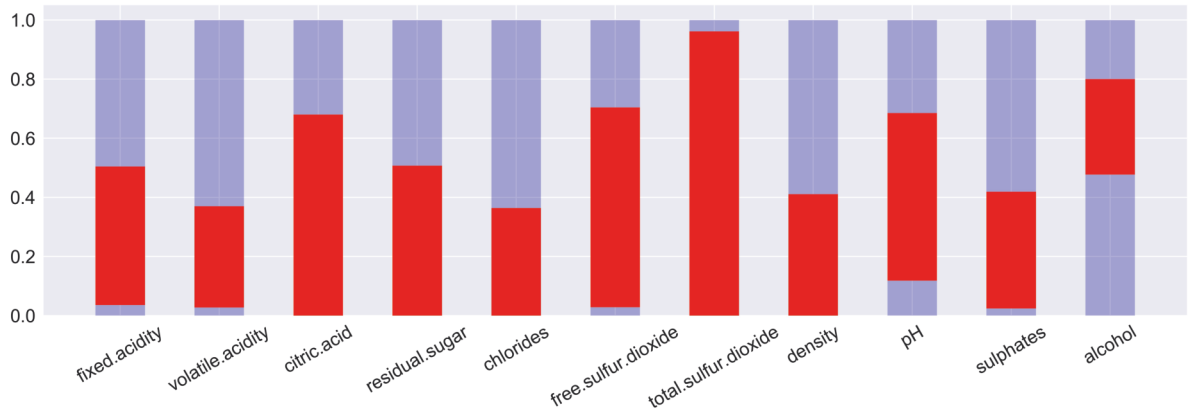


Figure .2:: Interval of the best scored wine from the random forest with 2 decision trees

This interval contains of 108 wines of the trainingset, 99 of them are good. If we try to predict the classes of the testset with this interval we receive the result below.

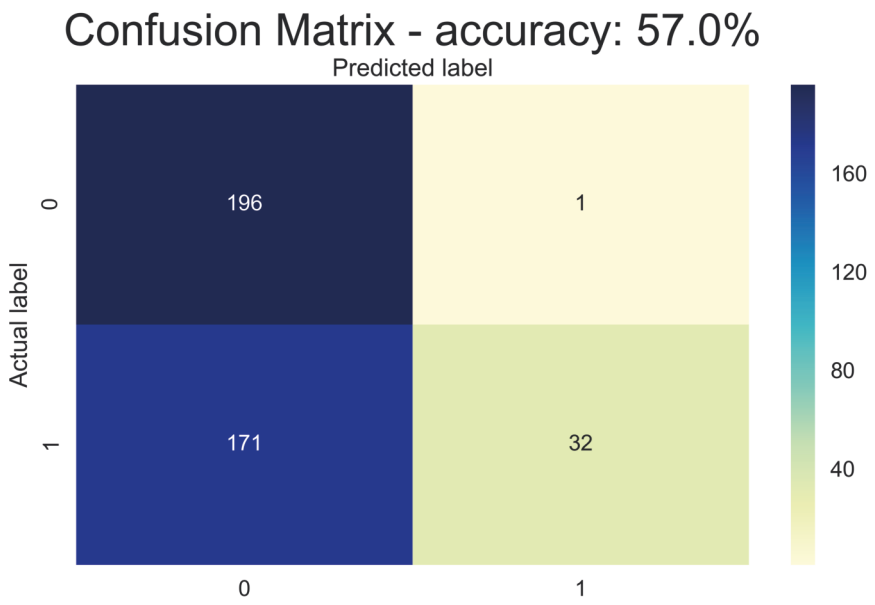


Figure .3:: Wines of the testset in the constructed interval

The next example is the random forest from Section 7.4 with 10 decision trees. The random forest delivers the following confusion matrix by predicting the classes of the trainingset:

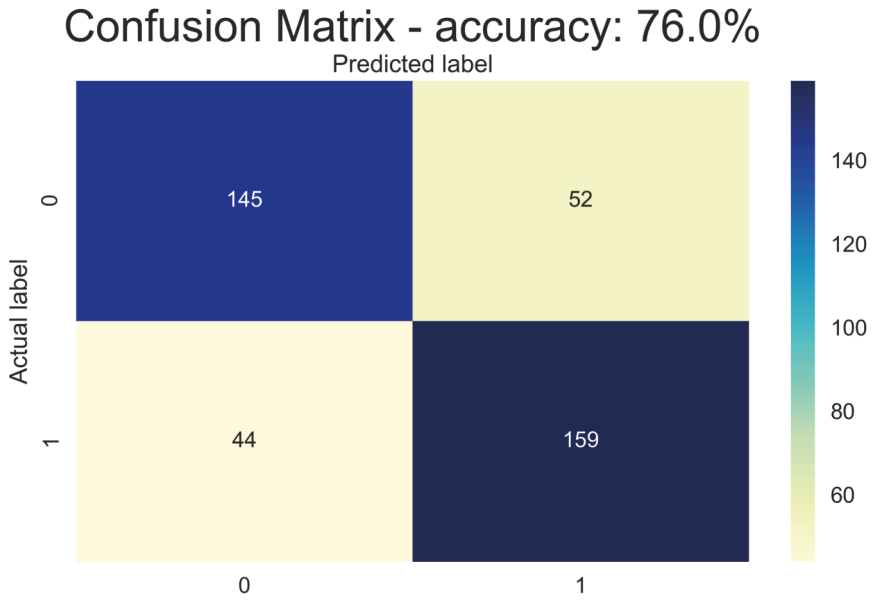


Figure .4:: Random forest build from 10 decision trees

The best scored wine of this random forest leads us to the following interval:

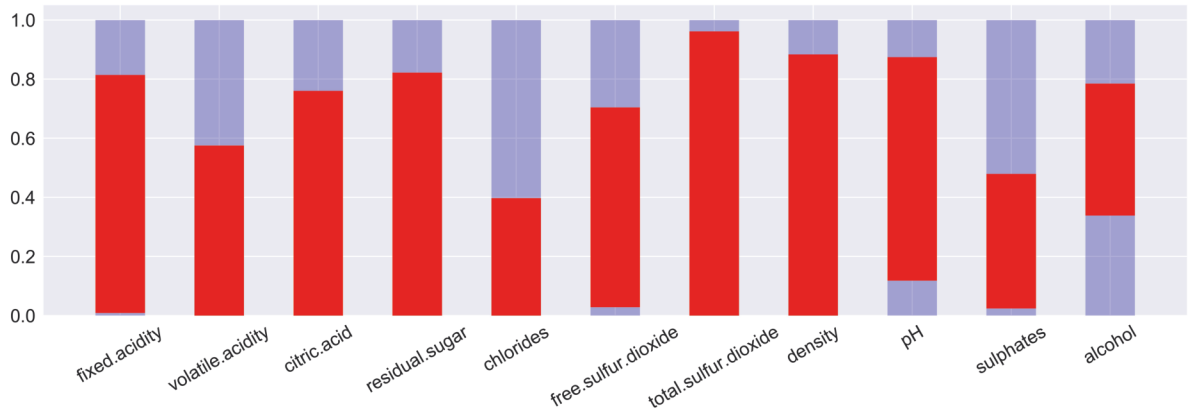


Figure .5:: Interval of the best scored wine from the random forest with 10 decision trees

This interval contains of 406 wines of the trainingset, 338 of them are good. If we try to predict the classes of the testset with this interval we receive the result below.

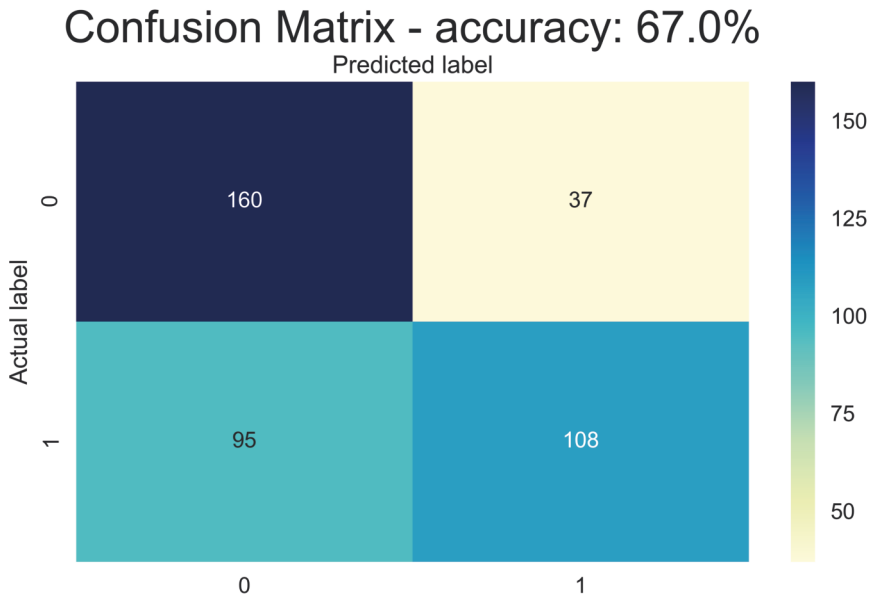


Figure .6:: Wines of the testset in the constructed interval

The last example is constructed by a random forest with 50 decision trees. Predicting the classes of the testset with it leads to the following results:

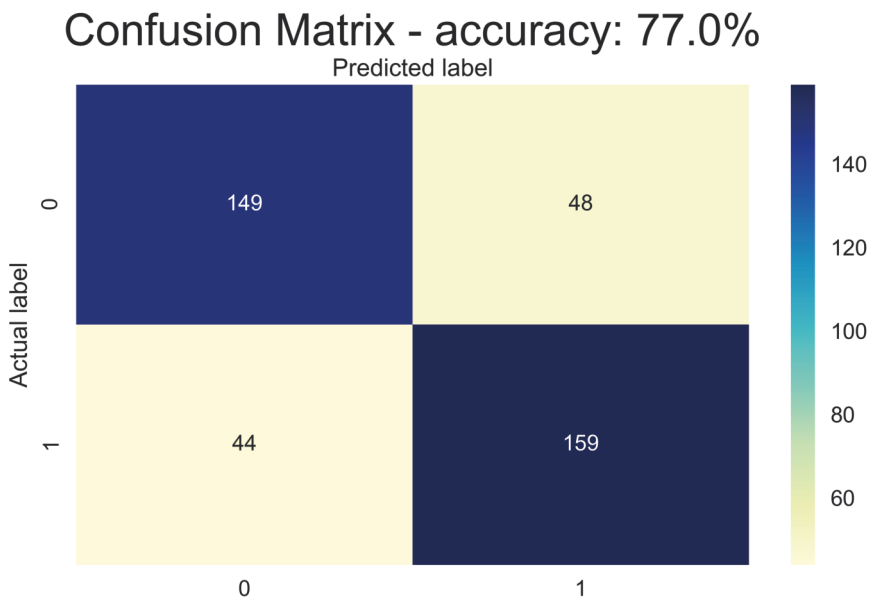


Figure .7:: Random forest build from 50 decision trees

The best scored wine of this random forest delivers us the interval shown below.

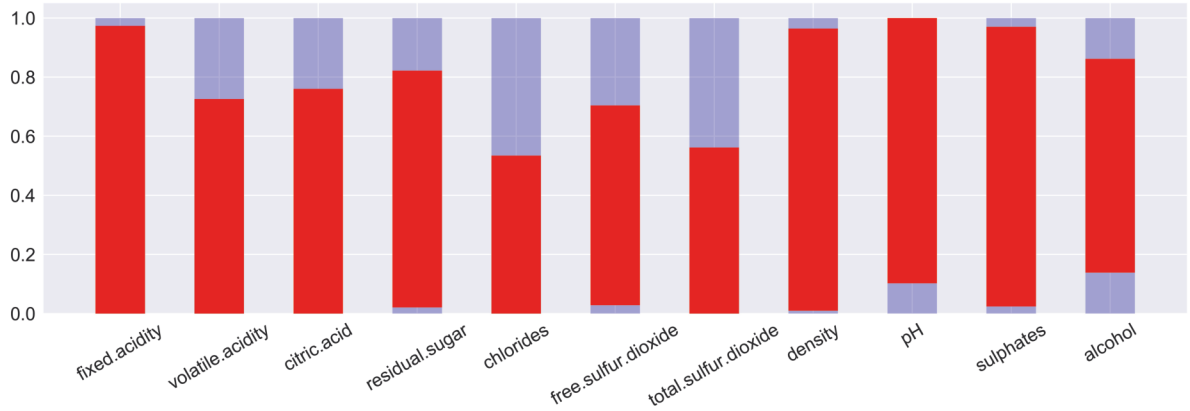


Figure .8:: Interval of the best scored wine from the random forest with 50 decision trees

This interval contains 1075 wines, 603 of them are good. A wine of the trainingset, which lies in this interval is thus with only a chance of 57,0% a good wine. If we use this interval to predict the classes of the testset it is obvious that the range of the attributes is too large because the interval contains many wines of the testset like the following confusion matrix shows.

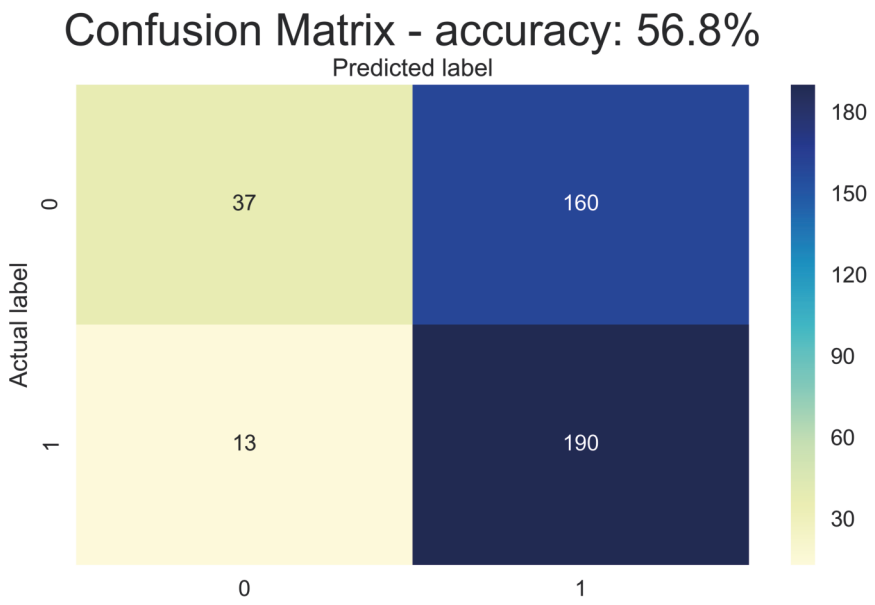


Figure .9:: Wines of the testset in the constructed interval

C: Get Cluster Pattern Intervals

This python code was used to create Figure 7.1, Figure 7.2, Figure 7.3, Figure 7.4, Figure 7.5, Figure 7.6, Figure 7.7, Figure 7.8, Figure 7.9, Figure 7.10, Figure 7.11, Figure 7.12, Figure 7.13

```
import pandas as pd
import numpy as np
import seaborn as sns
import random
import matplotlib
import sympy
import sklearn.tree
import csv


from numpy import *
from matplotlib import pyplot as plt
from sklearn import preprocessing
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics.pairwise import pairwise_distances
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from datetime import date
from itertools import *

from sklearn.cluster import KMeans
from sklearn.cluster import AffinityPropagation
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.cluster import SpectralClustering
from sklearn.cluster import DBSCAN
from sklearn.cluster import Birch
from sklearn.cluster import estimate_bandwidth
from sklearn.cluster import MeanShift
from sklearn.mixture import GaussianMixture
from sklearn.mixture import BayesianGaussianMixture

from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

from sklearn.tree import export_graphviz


import plotly
plotly.offline.init_notebook_mode()

import plotly.graph_objs as go

import operator
import copy


plt.style.use('seaborn')
random.seed(42)

def kMedoids(D, k, tmax=100):
    # determine dimensions of distance matrix D
    m, n = D.shape

    if k > n:
        raise Exception('too many medoids')

    # find a set of valid initial cluster medoid indices since we
    # can't seed different clusters with two points at the same location
    valid_medoid_inds = set(range(n))
    invalid_medoid_inds = set([])
    rs,cs = np.where(D==0)
    # the rows, cols must be shuffled because we will keep the first duplicate
                                            below
    index_shuf = list(range(len(rs)))
    np.random.shuffle(index_shuf)
    rs = rs[index_shuf]
    cs = cs[index_shuf]
    for r,c in zip(rs,cs):
        # if there are two points with a distance of 0...
        # keep the first one for cluster init
        if r < c and r not in invalid_medoid_inds:
            invalid_medoid_inds.add(c)
    valid_medoid_inds = list(valid_medoid_inds - invalid_medoid_inds)

    if k > len(valid_medoid_inds):
        raise Exception('too many medoids (after removing {} duplicate points)'.
                                                format(
            len(invalid_medoid_inds)))
```

```
    # randomly initialize an array of k medoid indices
    M = np.array(valid_medoid_inds)
    np.random.shuffle(M)
    M = np.sort(M[:k])

    # create a copy of the array of medoid indices
    Mnew = np.copy(M)

    # initialize a dictionary to represent clusters
    C = {}
    for t in range(tmax):
        # determine clusters, i. e. arrays of data indices
        J = np.argmin(D[:,M], axis=1)
        for kappa in range(k):
            C[kappa] = np.where(J==kappa)[0]
        # update cluster medoids
        for kappa in range(k):
            J = np.mean(D[np.ix_(C[kappa],C[kappa])],axis=1)
            j = np.argmin(J)
            Mnew[kappa] = C[kappa][j]
        np.sort(Mnew)
        # check for convergence
        if np.array_equal(M, Mnew):
            break
        M = np.copy(Mnew)
    else:
        # final update of cluster memberships
        J = np.argmin(D[:,M], axis=1)
        for kappa in range(k):
            C[kappa] = np.where(J==kappa)[0]

    # return results
    return M, C


# erstellen der Pattern Matrix
def get_pattern_matrices (M,label):

    if type(label) is dict:
        m = len(M[0])
        P_max = np.zeros([1,m],dtype=int)
        P_min = np.zeros([1,m],dtype=int)

        for lab in label:
            P_max_col = [0]
            P_max_col[0] = max(M[label[lab]][:,0])
            for col in range(1,m):
                P_max_col.append(max(M[label[lab]][:,col]))

            P_max = np.vstack([P_max,P_max_col])
```

```
    P_max = np.delete(P_max, (0), axis=0)


    for lab in label:
        P_min_col = [0]
        P_min_col[0] = min(M[label[lab]][:,0])
        for col in range(1,m):
            P_min_col.append(min(M[label[lab]][:,col]))

        P_min = np.vstack([P_min,P_min_col])

    P_min = np.delete(P_min, (0), axis=0)

else:

    n = len(M[0])
    #print(n)
    max_label = max(label)
    min_label = min(label)

    P_max = np.zeros([1,n],dtype=int)
    P_min = np.zeros([1,n],dtype=int)

    # erstes Maximum bestimmen an das die anderen angehÃd'ngt werden
    P_max_col = [0]
    P_max_col[0] = max(M[label == min_label][:,0])

    # erstes Minimum bestimmen an das die anderen angehÃd'ngt werden
    P_min_col = [0]
    P_min_col[0] = min(M[label == min_label][:,0])


    for l in range(max_label + 1):
        M_l = np.array(M[label==l])

        if M_l.size == 0:
            continue

        P_max_col = [0]
        P_max_col[0] = max(M_l[:,0])
        for col in range(1,n):
            P_max_col.append(max(M_l[:,col]))



        P_max = np.vstack([P_max,P_max_col])

    P_max = np.delete(P_max, (0), axis=0)
```

```
        for l in range(max_label + 1):
            M_l = np.array(M[label==l])

            if M_l.size == 0:
                continue

            P_min_col = [0]
            P_min_col[0] = min(M_l[:,0])
            for col in range(1,n):
                P_min_col.append(min(M_l[:,col]))



            P_min = np.vstack([P_min,P_min_col])

        P_min = np.delete(P_min, (0), axis=0)

    return P_max, P_min


def countcustomerinpattern(customer,P_min,P_max):
    pattern = len(P_min)
    attributes = len(P_min[0])
    customer_count = len(customer)
    counter = np.zeros(pattern)


    for p1 in range(pattern):
        for c in range(customer_count):
                for a in range(attributes):
                    if customer[c][a] < P_min[p1][a] or customer[c][a] > P_max[p1]
                                                                        [a]:
                        break
                    if a == (attributes-1):
                        counter[p1] = counter[p1] + 1

    return counter

def countcustomerspattern(customer,P_min,P_max):

    if P_min.size == 0:

        customer_count = len(customer)
        counter = np.zeros(customer_count)

        return counter


    else:
```

```
        #try:
            #pattern = len(P_min)
        #except:
        pattern = P_min.shape[0]

        #try:
            #attributes = len(P_min[0])
        #except:

        try:
            attributes = P_min.shape[1]

        except:
            attributes = P_min.shape[0]
            pattern = 0


        customer_count = len(customer)
        counter = np.zeros(customer_count)

        for c in range(customer_count):


            if pattern == 0:

                for p1 in range(pattern + 1):

                    for a in range(attributes):

                        if customer[c,a] < P_min[a] or customer[c,a] > P_max[a]:
                            break
                        if a == (attributes-1):
                            counter[c] = counter[c] + 1


            else:

                for p1 in range(pattern):


                    for a in range(attributes):

                        if customer[c,a] < P_min[p1,a] or customer[c,a] > P_max[p1
                                                                                ,a]:
                            break
                        if a == (attributes-1):
                            counter[c] = counter[c] + 1

        return counter
```

```
def finalpattern(customer,P_min_t,P_max_t):

    attributes = len(P_min_t[0])

    pattern_anz = len(P_min_t)

    P_max_final = np.zeros([1,attributes])
    P_min_final = np.zeros([1,attributes])

    counterc = countcustomerspattern(customer,P_min_t,P_max_t)

    #print(counterc)

    counterp_t = countcustomerinpattern(customer,P_min_t,P_max_t)

    #print("counterp_t: "  + str(counterp_t))

    for i in range(pattern_anz):


        try:
            minp = np.max(counterp_t[np.nonzero(counterp_t)])


        except:
            break


        next_pattern_min = P_min_t[np.where(counterp_t == minp)]
        next_pattern_max = P_max_t[np.where(counterp_t == minp)]

        next_pattern_min = np.mat(next_pattern_min[0,:])
        next_pattern_max = np.mat(next_pattern_max[0,:])



        counterc_p = countcustomerspattern(customer,next_pattern_min,
                                                    next_pattern_max)
        counterc_f = countcustomerspattern(customer,P_min_final,P_max_final)

        counterc_p[counterc_p > 0]=1
        counterc_f[counterc_f > 0]=1

        counterc = counterc_p - counterc_f
        counter = sum(i > 0 for i in counterc)
        #print("sdehfv: " + str(counter))

        if counter > 0:
            P_min_final = np.concatenate((P_min_final, next_pattern_min))
            P_max_final = np.concatenate((P_max_final, next_pattern_max))
```

```
        P_min_t = np.delete(P_min_t, np.where(counterp_t == minp)[0][0], 0)
        P_max_t = np.delete(P_max_t, np.where(counterp_t == minp)[0][0], 0)
        counterp_t = np.delete(counterp_t, np.where(counterp_t == minp)[0][0], 0)



        i = i + 1

    return P_min_final, P_max_final

def SelBest(arr:list, X:int)->list:
    '''
    returns the set of X configurations with shorter distance
    '''
    dx=np.argsort(arr)[:X]
    return arr[dx]

# Import
df = pd.read_excel (r'C:\Users\larsl\Desktop\Mathe\Python\red-wine-dataset\
                                        wineQualityReds01.xlsx')
df.dropna(inplace = True)


print(df.columns)

df.head()

df.hist(bins=15, color='steelblue', edgecolor='black', linewidth=1.0,
xlabelsize=8, ylabelsize=8, grid=False)

plt.tight_layout(rect=(0, 0, 1.0, 1.0))
plt.show()

pca = PCA(n_components=2)

principalComponents = pca.fit_transform(df)

principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2'])

finalDf = pd.concat([principalDf, df[['quality']]], axis = 1)

fig = plt.figure(figsize = (15,15))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
```

```
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

print('test')

targets = [0, 1]
colors = ['r', 'g']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['quality'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50
               , alpha = 0.3)
ax.legend(targets)
ax.grid()

X_total = df
y_test = df['quality']
y_test_total = np.array(y_test)
X_total = X_total.drop(['quality'],axis = 1)
colums_without_label = X_total.columns
X_total = np.array(X_total)

# Seperate good and bad
good = df[df.quality==1]
X = good.drop(['quality'],axis = 1)
X = np.array(X)

bad = df[df.quality==0]
X_bad = bad.drop(['quality'],axis = 1)
X_bad = np.array(X_bad)

# Histogramme fÃijr Merkmale erstellen
for column in df:
    plt.hist((good[column],bad[column]), 10, alpha=0.5, label=('good wine', 'bad
                                            wine'), color=('g','r') )
    lgd = plt.legend(bbox_to_anchor=(1.04,1), loc="center left", fontsize=20)
    plt.xticks(fontsize = 16)
    plt.yticks(fontsize = 16)
    plt.ylabel('Quantity', fontsize=20)
    plt.savefig('Histogram' + str(column) + '.jpg', dpi = 600, bbox_extra_artists=
                                            (lgd,), bbox_inches='tight')
    plt.show()

# Anzahl an optimalen Cluster bestimmen und Daten skalieren

mms = MinMaxScaler()
mms.fit(X_total)
X_bad_transformed = mms.transform(X_bad)
```

```
X_total_transformed = mms.transform(X_total)

print(y_test.shape)

X_transformed_gandb, X_test_transformed, y_train, y_test = train_test_split(
                                            X_total_transformed, y_test_total,
                                            test_size = 0.25, random_state = 4)

X_transformed = X_transformed_gandb[y_train == 1]

X_bad_transformed = X_transformed_gandb[y_train == 0]

X_test_transformed = np.array(X_test_transformed)
X_transformed = np.array(X_transformed)
X_bad_transformed = np.array(X_bad_transformed)

Sum_of_squared_distances = []
K = range(1,500,50)
for k in K:
    km = KMeans(n_clusters = k)
    km = km.fit(X_transformed)
    Sum_of_squared_distances.append(km.inertia_)

plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k')
plt.show()

acc_total_old = 0
imp_treshold = 0
k = 0
algo = 0
step = 0
step_treshold = 0.05

algo_names = [#'GaussianMixture (tied)' #69,5; 31
              #'k-Medoids (correlation)' #65,75; 15
              'k-Medoids (euclidean)' #66,5; 16
              ,'decision tree cluster (gini)'
              ,'k-Medoids (mahalanobis)' #66; 15
              ,'k-Means' #63,5 23
              ,'decision tree cluster (entropy)' #71,5; 34
              #,'MeanShift' #53,5 8
              #,'Agglomerative Clustering' #47,5 0
              #,'Agglomerative average Clustering'
              #,'Agglomerative ward Clustering'
              #,'Birch Clustering'
              #,'Spectral Clustering'
              #,'k-Medoids (cosine)' #66 25
```

```
                #,'k-Medoids (braycurtis)' #67 24
                #,'k-Medoids (canberra)' #68,25 24
                #,'Mini Batch k-Means' #66,5 17
                #,'GaussianMixture'
                #,'GaussianMixture (diag)'
                #,'GaussianMixture (spherical)'
                #,'Bayesian GaussianMixture'
                #,'Bayesian GaussianMixture (tied)'
                #,'Bayesian GaussianMixture (diag)'
                #'Bayesian GaussianMixture (spherical)'
               ]

algo_number = len(algo_names)
algo_counter = 0
algo_name = algo_names[algo_counter]

X_transformed_t = X_transformed
X_bad_transformed_t = X_bad_transformed
# untere und obere Schranke fÃijr Clusteranzahl eingeben
unten = 10
oben = 150
algo_attempts = 5
rounds_per_algo = 1

# Schranke fÃijr ratio in pattern
treshold_r = 0.7
treshold_r_temp = treshold_r
treshold_input = 0.7
treshold_d  = treshold_input
# Schranke fÃijr gute Beispiele in pattern
treshold_c = 100
P_min_dict = {}
P_max_dict = {}

counter_g_dict = {}
counter_b_dict = {}

ratio_dict = {}

pred_train = {}
pred_test = {}

attributes = len(X_transformed_t[0])

P_max_total = np.zeros([1,attributes])
P_min_total = np.zeros([1,attributes])

while k < rounds_per_algo and algo < algo_attempts and treshold_r_temp >=
                                        treshold_d and algo_counter < algo_number
                                        :
```

```
print("neuer Lauf k: " + str(k) + " algo: " + str(algo_name) + " " +str(algo)
                                            + " treshold: " + str(treshold_r_temp
                                            ))
n_clusters = random.randint(round(0.01*len(X_transformed)),round(0.1*len(
                                            X_transformed)))
print("noch nicht erkannt: " + str(len(X_transformed_t)))
print("Cluster: " + str(n_clusters))
seed = random.randint(1,10000)
features_tree = random.uniform(0.6, 1)

#76,25 18
# Auswahl des Cluster Algorithmus
if algo_name == 'k-Medoids (correlation)' : #10 steps 70,625% A0P1=12; neu 20
                                            70,9
    treshold_d  = treshold_input
    D = pairwise_distances(X_transformed, metric= 'correlation')
    M, cluster = kMedoids(D,n_clusters)

elif algo_name == 'k-Medoids (euclidean)': #10 steps 66,25% A0P1=12; neu 20 69
                                            ,7
    treshold_d  = treshold_input
    D = pairwise_distances(X_transformed, metric='euclidean')
    M, cluster = kMedoids(D,n_clusters)

elif algo_name == 'decision tree cluster (gini)': #10 steps 70% A0P1=24; neu
                                            20 75,6
    treshold_d  = 0.7
    clf = sklearn.tree.DecisionTreeClassifier(max_features = features_tree,
                                                min_samples_leaf = treshold_c)
    clf.fit(X_transformed_gandb,y_train)
    cluster_total_tree = clf.tree_.apply(np.asfortranarray(X_transformed_gandb
                                                .astype(sklearn.tree._tree.DTYPE)
                                                ))
    cluster = cluster_total_tree[y_train == 1]

elif algo_name == 'decision tree cluster (entropy)': #10 steps 71,85% A0P1=27;
                                             neu 20 75,6
    treshold_d  = 0.7
    clf = sklearn.tree.DecisionTreeClassifier(criterion = 'entropy',
                                                max_features = features_tree,
                                                min_samples_leaf = treshold_c)
    clf.fit(X_transformed_gandb,y_train)
    cluster_total_tree = clf.tree_.apply(np.asfortranarray(X_transformed_gandb
                                                .astype(sklearn.tree._tree.DTYPE)
                                                ))
    cluster = cluster_total_tree[y_train == 1]

elif algo_name == 'k-Medoids (mahalanobis)': #10 steps 70,31% A0P1=21; neu 20
                                            72,8 war aber bei 75
    treshold_d  = treshold_input
```

```
        D = pairwise_distances(X_transformed, metric='mahalanobis')
        M, cluster = kMedoids(D,n_clusters)

    elif algo_name == 'k-Means': #10 steps 69,375% A0P1=18; neu 20 71,5
        treshold_d  = treshold_input
        cluster = KMeans(n_clusters, random_state = seed, init = 'random').
                                                fit_predict(X_transformed)

    elif algo_name == 'MeanShift': #10 steps 69,375% A0P1=18; ging nicht
        treshold_d  = treshold_input
        bandwidth = random.uniform(0, 0.5) #estimate_bandwidth(X, quantile=0.2,
                                                random_state = seed)
        print('bandwidth', bandwidth)
        cluster = MeanShift(bandwidth = bandwidth, cluster_all = 'true').
                                                fit_predict(X_transformed)

    elif algo_name == 'Agglomerative Clustering': #10 steps 57,185% A0P1=3; neu 66
        treshold_d  = treshold_input
        n_clusters = n_clusters * 2
        cluster = AgglomerativeClustering(n_clusters).fit_predict(X_transformed)

    elif algo_name == 'Agglomerative average Clustering': #10 steps 55,625% A0P1=0
                                            ; neu 20 70
        treshold_d  = treshold_input
        n_clusters = random.randint(round(0.01*len(X_transformed)),round(0.3*len(
                                                X_transformed)))
        cluster = AgglomerativeClustering(n_clusters, linkage='average',
                                                pooling_func='deprecated').
                                                fit_predict(X_transformed)

    elif algo_name == 'Agglomerative ward Clustering': #10 steps 55,625% A0P1=0;
                                            neu 20 70
        treshold_d  = treshold_input
        n_clusters = random.randint(round(0.1*len(X_transformed)),round(0.3*len(
                                                X_transformed)))
        cluster = AgglomerativeClustering(n_clusters, linkage='ward', pooling_func
                                                ='deprecated').fit_predict(
                                                X_transformed)

    elif algo_name == 'Birch Clustering' : #10 steps 53,125% A0P1=1 neu 20 67 (
                                            treshold auf 70)
        treshold_d  = treshold_input
        n_clusters = random.randint(round(0.2*len(X_transformed)),round(0.6*len(
                                                X_transformed)))
        cluster = Birch(n_clusters = n_clusters, threshold = 0.2).fit_predict(
                                                X_transformed)

    elif algo_name == 'Spectral Clustering': #10 steps 52,821% A0P1=1; neu 20 72,5
        treshold_d  = treshold_input
        cluster = SpectralClustering(n_clusters, random_state = seed).fit_predict(
```

```
                                                X_transformed)

    elif algo_name == 'k-Medoids (cosine)': #10 steps 70,625% A0P1=19; neu 20 72
        treshold_d  = treshold_input
        D = pairwise_distances(X_transformed, metric='cosine')
        M, cluster = kMedoids(D,n_clusters)

    elif algo_name == 'k-Medoids (braycurtis)': #10 steps 69,6785% A0P1=19; neu 20
                                                 72
        treshold_d  = treshold_input
        D = pairwise_distances(X_transformed, metric= 'braycurtis')
        M, cluster = kMedoids(D,n_clusters)

    elif algo_name == 'k-Medoids (canberra)': #10 steps 66,875% A0P1=21; neu 20 70
                                               ,6
        treshold_d  = treshold_input
        D = pairwise_distances(X_transformed, metric= 'canberra')
        M, cluster = kMedoids(D,n_clusters)

    elif algo_name == 'k-Medoids (dice)':
        treshold_d  = treshold_input
        D = pairwise_distances(X_transformed, metric= 'dice')
        M, cluster = kMedoids(D,n_clusters)

    elif algo_name == 'Mini Batch k-Means': #10 steps 70,3125% A0P1=14; neu 20 71,
                                             8
        treshold_d  = treshold_input
        cluster = MiniBatchKMeans(n_clusters, random_state = seed, init = 'random'
                                                , batch_size = 100).fit_predict(
                                                X_transformed)

    elif algo_name == 'GaussianMixture': #10 steps 68,4% A0P1=14; neu 20 70
        treshold_d  = treshold_input
        cluster = GaussianMixture(n_clusters, init_params='random').fit_predict(
                                                X_transformed)

    elif algo_name == 'GaussianMixture (tied)' : #10 steps 66,25% A0P1=8; 25 steps
                                                  76,5% 45; neu 20 76,8
        treshold_d  = treshold_input
        n_clusters = random.randint(round(0.01*len(X_transformed)),round(0.4*len(
                                                X_transformed)))
        cluster = GaussianMixture(n_clusters, covariance_type = 'tied',
                                                init_params='random').fit_predict
                                                (X_transformed)

    elif algo_name == 'GaussianMixture (diag)': #10 steps 65,625% A0P1=10; neu 20
                                                 70
        treshold_d  = treshold_input
        n_clusters = random.randint(round(0.01*len(X_transformed)),round(0.1*len(
                                                X_transformed)))
```

```
    cluster = GaussianMixture(n_clusters, covariance_type = 'diag',
                                            init_params='random').fit_predict
                                            (X_transformed)

elif algo_name == 'GaussianMixture (spherical)': #10 steps 65,9375% A0P1=7;
                                        neu 20 72,2
    treshold_d  = treshold_input
    cluster = GaussianMixture(n_clusters, covariance_type = 'spherical',
                                            init_params='random').fit_predict
                                            (X_transformed)

elif algo_name == 'Bayesian GaussianMixture': #10 steps 63,4375 A0P1 = 4; neu
                                        20 72,8
    treshold_d  = treshold_input
    cluster = BayesianGaussianMixture(n_clusters, init_params = 'random').
                                            fit_predict(X_transformed)

elif algo_name == 'Bayesian GaussianMixture (tied)': #10 steps 67,5% A0P1 = 6;
                                         neu 20 70
    treshold_d  = treshold_input
    cluster = BayesianGaussianMixture(n_clusters, covariance_type = 'tied').
                                            fit_predict(X_transformed)

elif algo_name == 'Bayesian GaussianMixture (diag)': #10 steps 60,625% A0P1 =
                                        3; neu 20 68,4
    treshold_d  = treshold_input
    cluster = BayesianGaussianMixture(n_clusters, covariance_type = 'diag').
                                            fit_predict(X_transformed)

elif algo_name == 'Bayesian GaussianMixture (spherical)': #10 steps 53,75%
                                        A0P1= 4; neu 20 64,3
    treshold_d  = treshold_input
    cluster = BayesianGaussianMixture(n_clusters, covariance_type = 'spherical
                                            ').fit_predict(X_transformed)



P_max,P_min = get_pattern_matrices(X_transformed,cluster)

# Berechnung wie viele Kunden in Pattern
counter_g_tmp = countcustomerinpattern(X_transformed_t,P_min,P_max)
counter_b_tmp = countcustomerinpattern(X_bad_transformed_t,P_min,P_max)


ratio_tmp = np.divide(counter_g_tmp,(counter_g_tmp + counter_b_tmp), out=np.
                                        zeros_like(counter_g_tmp), where =
                                        counter_g_tmp!=0)


#plt.hist(ratio,bins=10, color='b', label = 'pattern')
```

```
#plt.ylabel('quantity', fontsize = 13)
#plt.xlabel('', fontsize = 13)
#plt.title('Histogram ' + str(algo_name)+ ' pattern ' + str(step), fontsize =
                                        20)
#plt.legend(frameon=True,shadow=True, fancybox=True)
#plt.savefig('Histogram ' + str(algo_name) + ' step ' + str(step) + str(
                                        n_clusters) + '.png', dpi=600)
#plt.show()


P_max_t = P_max[np.where((ratio_tmp > treshold_r_temp) & (counter_g_tmp >
                                        treshold_c))]
P_min_t = P_min[np.where((ratio_tmp > treshold_r_temp) & (counter_g_tmp >
                                        treshold_c))]

counter_g = counter_g_tmp[np.where((ratio_tmp > treshold_r_temp) & (
                                        counter_g_tmp > treshold_c))]
counter_b = counter_b_tmp[np.where((ratio_tmp > treshold_r_temp) & (
                                        counter_g_tmp > treshold_c))]

ratio = ratio_tmp[np.where((ratio_tmp > treshold_r_temp) & (counter_g_tmp >
                                        treshold_c))]




counterc_p = countcustomerspattern(X_transformed_t,P_min_t,P_max_t)
counterc_n = countcustomerspattern(X_bad_transformed_t,P_min_t,P_max_t)

X_transformed_t = np.delete(X_transformed_t, np.where(counterc_p > 0), 0)
X_bad_transformed_t = np.delete(X_bad_transformed_t, np.where(counterc_n > 0),
                                         0)


if P_max_t.size == 0:
    if k == rounds_per_algo - 1:
        k = 0
        step = step + 1
        if treshold_r_temp - step_treshold < treshold_d:

            algo_name_and_attempt = str(algo_name) + str(algo)

            P_min_dict[algo_name_and_attempt] = P_min_total[1:,:]
            P_max_dict[algo_name_and_attempt] = P_max_total[1:,:]

            ratio_dict[algo_name_and_attempt] = ratio

            counter_g_dict[algo_name_and_attempt] = counter_g
```

```
                counter_b_dict[algo_name_and_attempt] = counter_b

                try:
                    pred_train[algo_name_and_attempt] = y_pred_count_total
                    pred_test[algo_name_and_attempt] = y_pred_count_test

                except:
                    pred_train[algo_name_and_attempt] = np.zeros(len(P_min_total))
                    pred_test[algo_name_and_attempt] = np.zeros(len(P_min_total))



                P_min_total = np.zeros([1,attributes])
                P_max_total = np.zeros([1,attributes])



                if algo == algo_attempts - 1:

                    algo = 0
                    algo_counter = algo_counter + 1
                    algo_name = algo_names[algo_counter]

                else:
                    algo = algo + 1

                treshold_r_temp = treshold_r
                X_transformed_t = X_transformed

            else:
                treshold_r_temp = treshold_r_temp - step_treshold

        else:
            k = k + 1

        continue

    y_pred_count = countcustomerspattern(X_total_transformed,P_min_t,P_max_t)
    y_pred = y_pred_count
    y_pred[y_pred > 0]=1

    #acc = metrics.accuracy_score(y_test, y_pred)
    #print("accuracy: " + str(acc))


    #cnf_matrix = metrics.confusion_matrix(y_test, y_pred)

    #class_names=[0,1] # name  of classes
    #fig, ax = plt.subplots()
    #tick_marks = np.arange(len(class_names))
```

```
#plt.xticks(tick_marks, class_names)
#plt.yticks(tick_marks, class_names)

# create heatmap
#sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
#ax.xaxis.set_label_position("top")
#plt.tight_layout()
#plt.title('Confusion matrix ' + str(algo_name) + ' step: ' + str(step) + '
                                        cluster: ' + str(n_clusters), y=1.1,
                                        fontsize =20)
#plt.ylabel('Actual label')
#plt.xlabel('Predicted label')
#fig.tight_layout()
#fig.savefig('confusion matrix' + str(algo_name) + ' step ' + str(step) + '
                                        cluster: ' + str(n_clusters) + '.png
                                        ', dpi=600)
#plt.show()


print("P_total:")

if P_max_t.size:
    print("pattern aktuell gesamt: " + str(P_max_total.size))
    print("jetzt neu dazu: " + str(P_max_t.size))
    P_max_total = np.concatenate((P_max_total, P_max_t))
    P_min_total = np.concatenate((P_min_total, P_min_t))


y_pred_count_total = countcustomerspattern(X_transformed_gandb,P_min_total,
                                        P_max_total)
y_pred_total = y_pred_count_total
y_pred_total[y_pred_total>0]=1

acc_total = metrics.accuracy_score(y_train, y_pred_total)
print("accuracy auf Trainingsdaten: " + str(acc_total))

cnf_matrix = metrics.confusion_matrix(y_train, y_pred_total)

class_names=[0,1] # name  of classes
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

#create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix total step: ' + str(step), y=1.1, fontsize =20)
plt.ylabel('Actual label')
```

```
plt.xlabel('Predicted label')
fig.tight_layout()
#fig.savefig('confusion total ' + 'step ' + str(step) + '.png', dpi=600)
plt.show()

improvement = acc_total - acc_total_old
print("improvement Training: " + str(improvement))
#print("imp_treshold: " + str(imp_treshold))
acc_total_old = acc_total



print(' ')
print('Bewertung Testset')
print(' ')

#Bewertung Testset
y_pred_count_test = countcustomerspattern(X_test_transformed,P_min_total,
                                          P_max_total)
y_pred_test = y_pred_count_test
y_pred_test[y_pred_test < 1]=0
y_pred_test[y_pred_test > 0]=1



acc_test = metrics.accuracy_score(y_test, y_pred_test)
print("accuracy Testset: " + str(acc_test))

cnf_matrix = metrics.confusion_matrix(y_test, y_pred_test)

class_names=[0,1] # name  of classes
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

# create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap = "YlGnBu" ,fmt = 'g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix test step: ' + str(step), y = 1.1, fontsize = 20)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
fig.tight_layout()
#fig.savefig('confusion total ' + 'step ' + str(step) + '.png', dpi=600)
plt.show()

print('Ende Bewertung Testset')
```

```
if improvement > imp_treshold:

    print('cwndcoin')

    if k == rounds_per_algo - 1:
        k = 0

        # Einblenden, falls dict genutzt werden soll. Ausgeblendet in letzten
                                                Eintrag in P_min_dict alle
                                                pattern.
        #P_max_total = np.zeros([1,attributes])
        #P_min_total = np.zeros([1,attributes])
        #print('da!!!!!!!!!!!!!!!!!!')

        if treshold_r_temp - step_treshold < treshold_d:

            algo_name_and_attempt = str(algo_name) + str(algo)

            print('cwndcoin')
            print(algo_name_and_attempt)

            P_min_dict[algo_name_and_attempt] = P_min_total[1:,:]
            P_max_dict[algo_name_and_attempt] = P_max_total[1:,:]

            ratio_dict[algo_name_and_attempt] = ratio

            counter_g_dict[algo_name_and_attempt] = counter_g
            counter_b_dict[algo_name_and_attempt] = counter_b

            #P_min_total = np.zeros([1,attributes])
            #P_max_total = np.zeros([1,attributes])

            try:
                pred_train[algo_name_and_attempt] = y_pred_count_total
                pred_test[algo_name_and_attempt] = y_pred_count_test

            except:
                pred_train[algo_name_and_attempt] = np.zeros(len(P_min_total))
                pred_test[algo_name_and_attempt] = np.zeros(len(P_min_total))


            if algo == algo_attempts - 1:

                algo = 0
                algo_counter = algo_counter + 1
                algo_name = algo_names[algo_counter]

            else:
                algo = algo + 1
```

```
                X_transformed_t = X_transformed
                treshold_r_temp = treshold_r


            else:
                treshold_r_temp = treshold_r_temp - step_treshold

            print("TRESHOLD: " + str(treshold_r_temp))
        else:
            k = k + 1

    else:
        k = 0

    step = step + 1

#np.set_printoptions(suppress=True)
#print("Pmax:")
#print(P_max)
#print("Pmin:")
#print(P_min)
#print("P_max_total")
#print(P_max_total)
#print("P_min_total")
#print(P_min_total)


for d in P_min_dict:
    if len(ratio_dict[d]) == 0:
        del ratio_dict[d]



print(ratio_dict)

import copy

anz_p = 5
ratio_max = np.zeros([anz_p])
ratio_argmax = np.zeros([anz_p], dtype= int)
ratio_max_tmp = 0

alg = np.zeros([anz_p],dtype=object)
ratio_dict_tmp = {}
ratio_dict_tmp = copy.deepcopy(ratio_dict)

for i in range(anz_p):

    for d in ratio_dict_tmp:
        try:
```

```
            ratio_max_tmp = max(ratio_dict_tmp[d])
            ratio_argmax_tmp = np.argmax(ratio_dict_tmp[d])

        except:
            ratio_max_tmp = 0



        #print(ratio_argmax_tmp)

        #print(ratio_max_tmp)

        if ratio_max_tmp > ratio_max[i]:
            alg_tmp = str(d)
            alg[i] = alg_tmp
            ratio_max[i] = ratio_max_tmp
            ratio_argmax[i] = ratio_argmax_tmp

    ratio_dict_tmp[alg[i]][ratio_argmax[i]] = 0


#print(d)
#print(alg)
print(ratio_dict_tmp)

#print(ratio_dict_tmp)

#print(ratio_dict)

m = len(P_min_dict[alg[0]][0])
P_min_final = np.zeros([1,m],dtype=int)
P_max_final = np.zeros([1,m],dtype=int)

print(P_min_final)




for i in range(anz_p):

    print('Algorithmus:')
    print(str(alg[i]))
    min_examples = X_transformed_gandb.max(axis = 0)
    print('gute Weine:')
    print(counter_g_dict[alg[i]][ratio_argmax[i]])
    print('schlechte Weine')
    print(counter_b_dict[alg[i]][ratio_argmax[i]])


    fig, ax = plt.subplots(figsize=(18,5))
```

```
plt.bar(colums_without_label, 1, width = 0.5, color = 'b', alpha=.3)

#for i in range(1):
    #if ratio[i] > 0.5:


plt.bar(colums_without_label, P_min_dict[alg[i]][ratio_argmax[i]], width = 0.5
                                        , color = 'b', alpha = 0)
plt.bar(colums_without_label, P_max_dict[alg[i]][ratio_argmax[i]] - P_min_dict
                                        [alg[i]][ratio_argmax[i]], width = 0.
                                        5, color = 'r',alpha = 1, bottom =
                                        P_min_dict[alg[i]][ratio_argmax[i]] )

plt.xticks(colums_without_label, rotation = 30, fontsize = 16)
plt.yticks(fontsize = 16)
plt.savefig('best pattern' + str(alg[i])+ '_' + str(counter_g_dict[alg[i]][
                                        ratio_argmax[i]]) + '_' +str(
                                        counter_b_dict[alg[i]][ratio_argmax[i
                                        ]]) + '.jpg', dpi = 600, bbox_inches=
                                        'tight')
plt.show()

#Bewertung Testset
y_pred_count_test = countcustomerspattern(X_test_transformed,P_min_dict[alg[i]
                                        ][ratio_argmax[i]],P_max_dict[alg[i]]
                                        [ratio_argmax[i]])
y_pred_test = y_pred_count_test
y_pred_test[y_pred_test < 1]=0
y_pred_test[y_pred_test > 0]=1



acc_test = metrics.accuracy_score(y_test, y_pred_test)
print("accuracy Testset: " + str(acc_test))

cnf_matrix = metrics.confusion_matrix(y_test, y_pred_test)

# name  of classes
class_names=[0,1]

fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

acc_tree = metrics.accuracy_score(y_test, y_pred_test)

acc_tree = acc_tree * 100
# create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu", fmt='g')
```

```
    ax.xaxis.set_label_position("top")
    plt.tight_layout()
    plt.title('Confusion Matrix - accuracy: '+str(acc_tree.round(1)) + '%', y=1.1,
                                                fontsize =20)
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')
    fig.tight_layout()
    fig.savefig('confusion matrix' + str(alg[i]) + '.png', dpi=600)
    plt.show()

    P_min_final = np.vstack([P_min_final,P_min_dict[alg[i]][ratio_argmax[i]]])
    P_max_final = np.vstack([P_max_final,P_max_dict[alg[i]][ratio_argmax[i]]])


P_min_final = np.delete(P_min_final, (0), axis=0)
P_max_final = np.delete(P_max_final, (0), axis=0)


#Bewertung Testset
y_pred_count_test = countcustomerspattern(X_test_transformed,P_min_final,
                                          P_max_final)
y_pred_test = y_pred_count_test
y_pred_test[y_pred_test < 1]=0
y_pred_test[y_pred_test > 0]=1



acc_test = metrics.accuracy_score(y_test, y_pred_test)
print("accuracy Testset: " + str(acc_test))

cnf_matrix = metrics.confusion_matrix(y_test, y_pred_test)

# name  of classes
class_names=[0,1]

fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

acc_tree = metrics.accuracy_score(y_test, y_pred_test)

acc_tree = acc_tree * 100
# create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu", fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion Matrix - accuracy: '+str(acc_tree.round(1)) + '%', y=1.1,
                                          fontsize =20)
plt.ylabel('Actual label')
```

```
plt.xlabel('Predicted label')
fig.tight_layout()
fig.savefig('confusion total cluster pattern' + '.png', dpi=600)
plt.show()
```

D: Intervalls of Single Attributes

In Section 8.1 we build a model to predict classes of red wines just on the attribute alcohol. In this section we present the models build from the other attributes. In some cases fewer intervals were found that met our criterias (more than 100 wines in the intervall and a ratio of at least 70% good wines in it).

D.1: Volatile Acidity

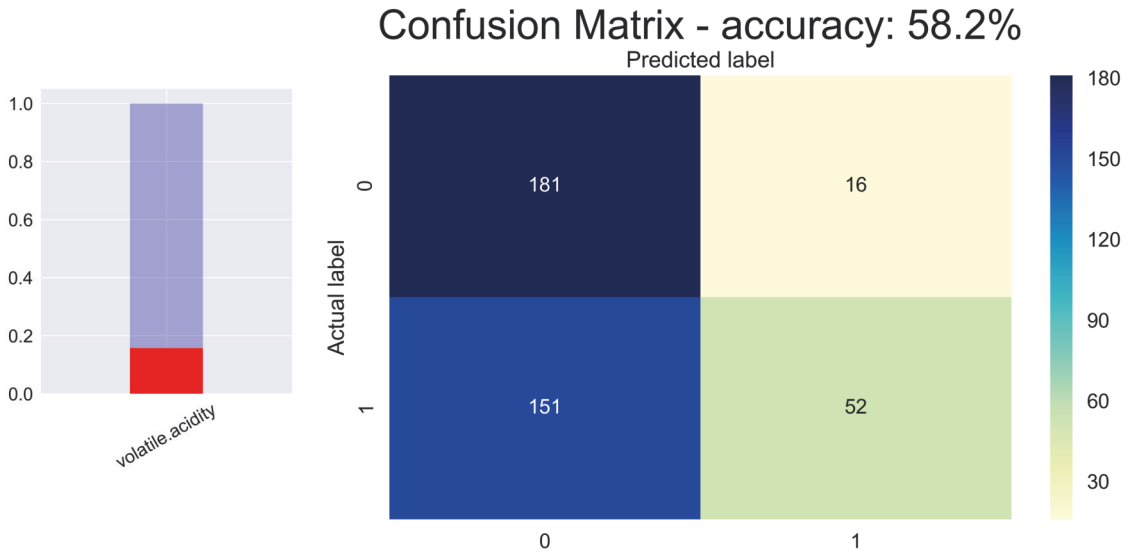


Figure .10:: Interval 1 (k-medoids - mahalanobis): 176 wines, 166 good and 10 bad

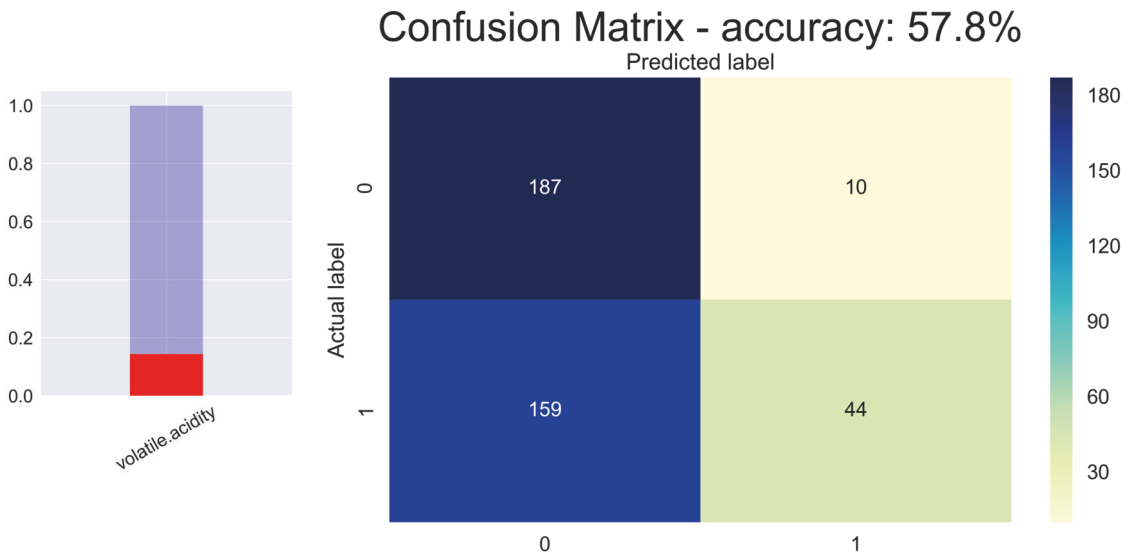


Figure .11:: Interval 2 (k-medoids - mahalanobis): 166 wines, 138 good and 28 bad

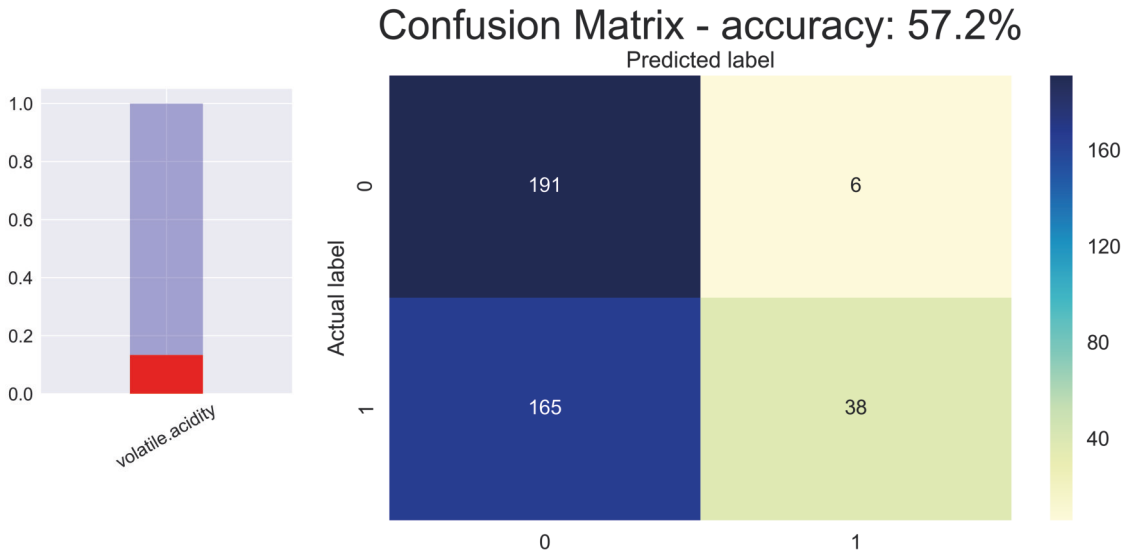


Figure .12:: Interval 3 (decision tree - entropy): 166 wines, 138 good and 28 bad

Combining these 3 intervals to predict the test set we receive the following result.

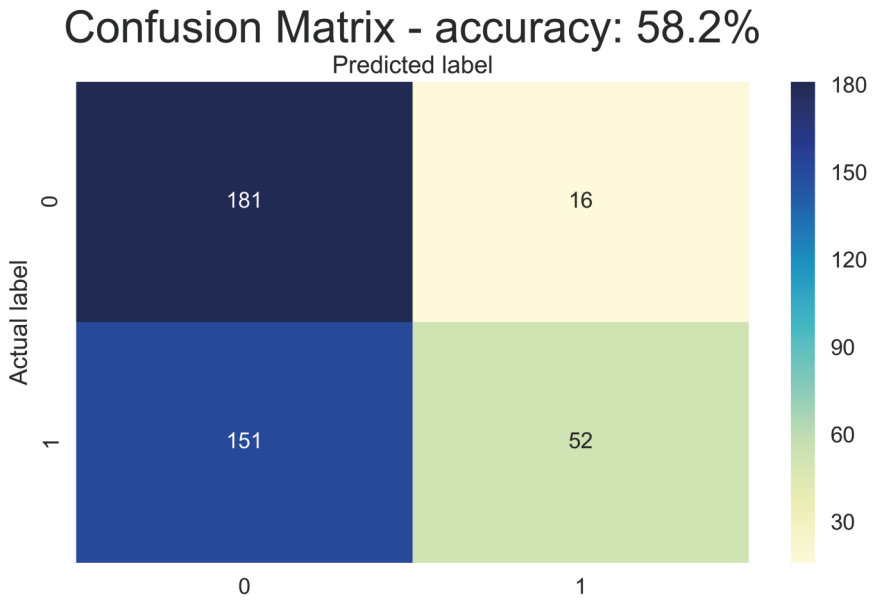


Figure .13:: Predicting the test set with the 3 intervals

D.2: Sulphates

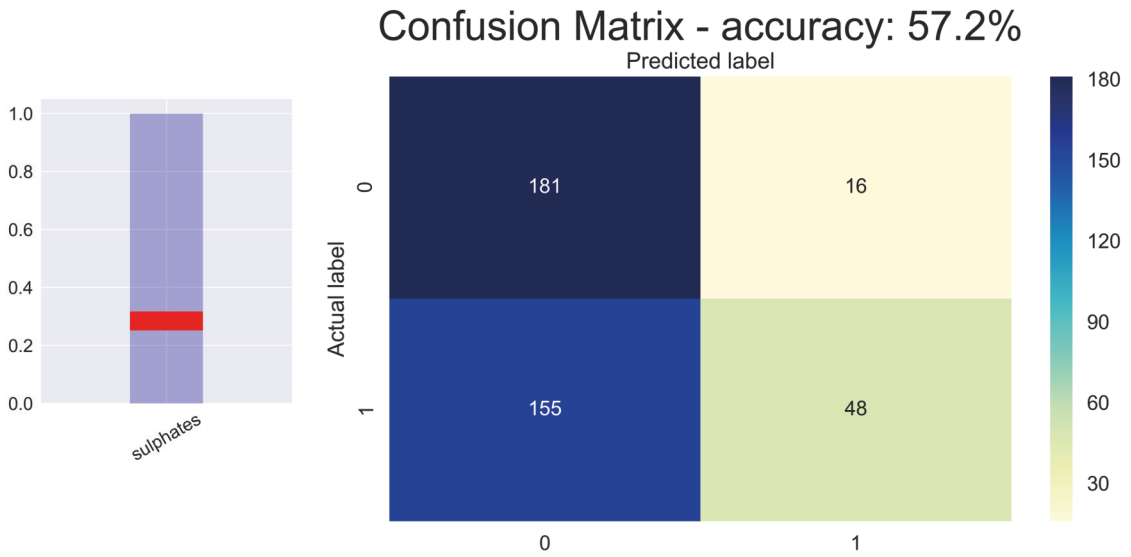


Figure .14:: Interval 1 (decision tree - entropy): 132 wines, 132 good and 0 bad

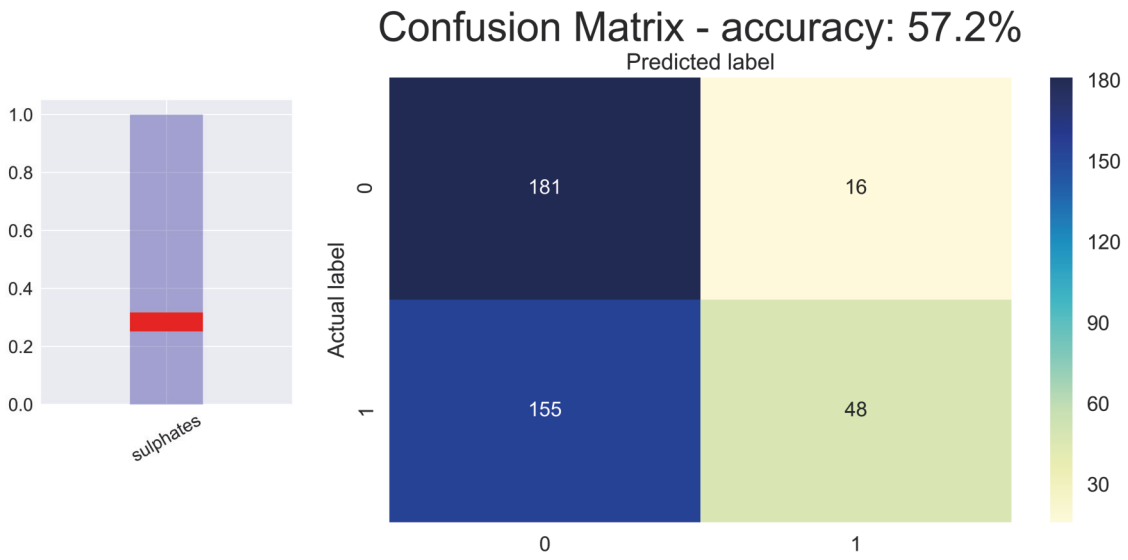


Figure .15:: Interval 2 (decision tree - entropy): 159 wines, 132 good and 27 bad

Combining these 2 intervals to predict the test set we receive the following result.

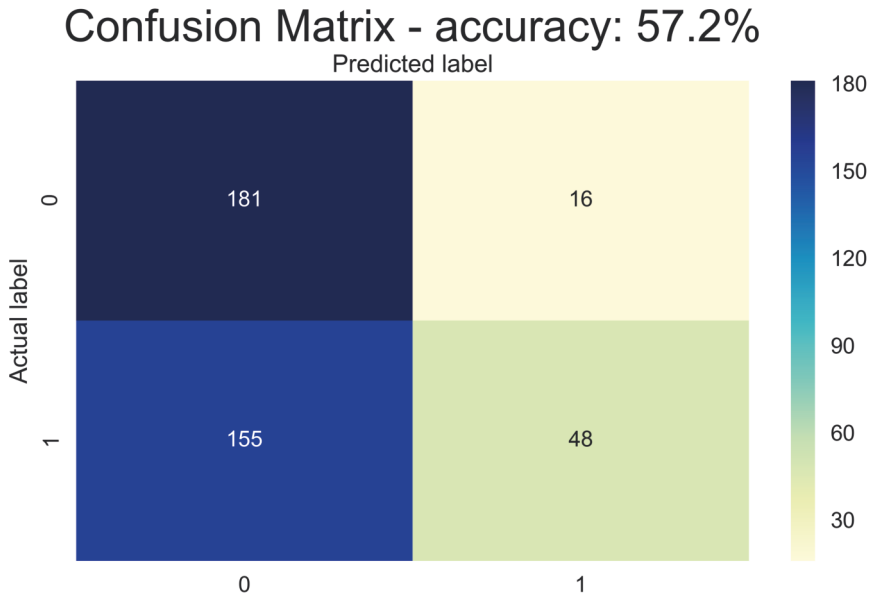


Figure .16:: Predicting the test set with the 2 intervals

D.3: Chlorides

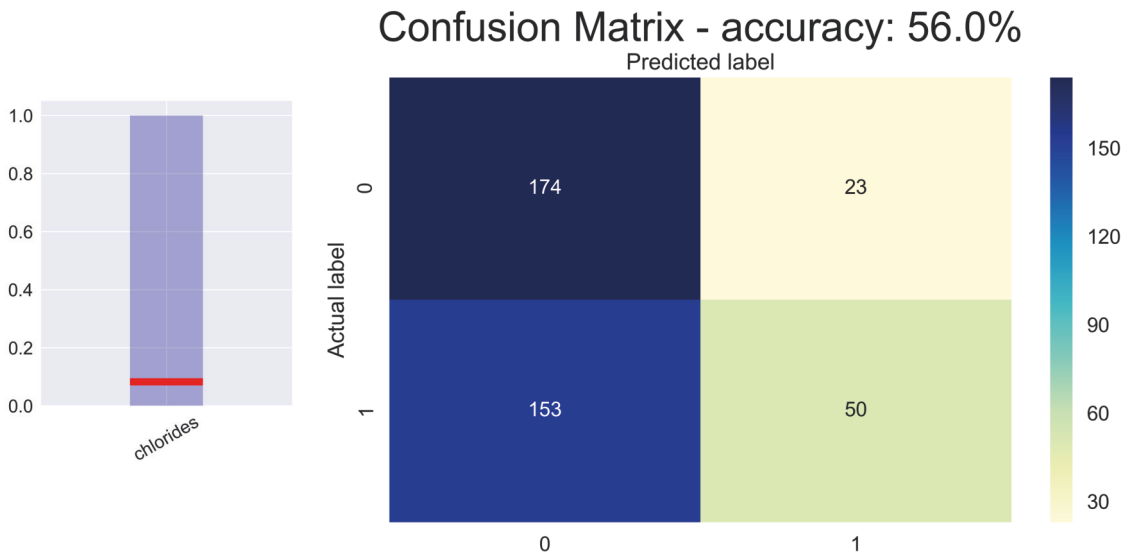


Figure .17:: Interval 1 (decision tree - entropy): 179 wines, 156 good and 23 bad

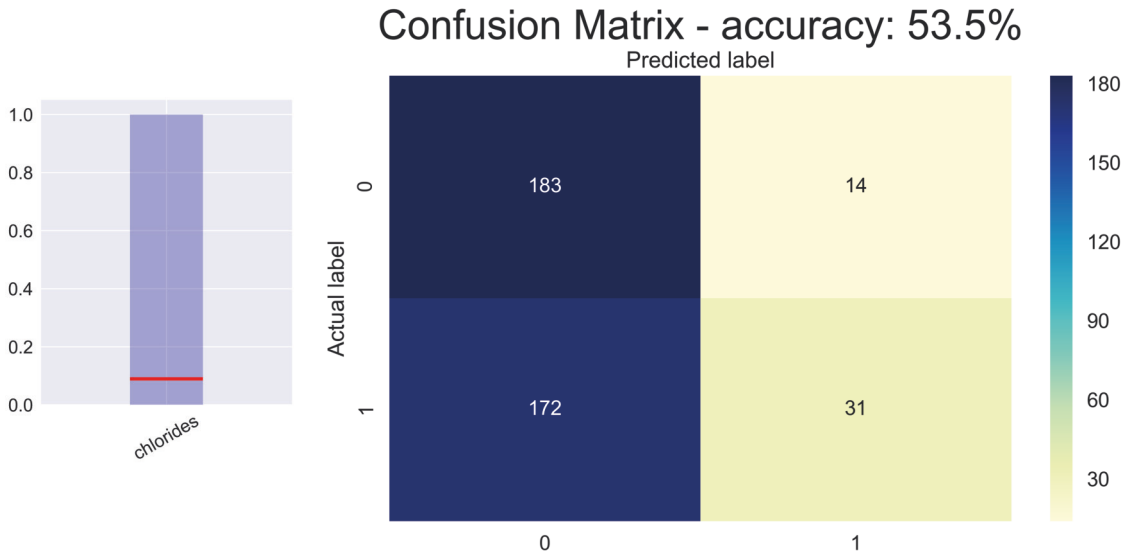


Figure .18:: Interval 2 (k-medoids - mahalanobis): 153 wines, 109 good and 45 bad

Combining these 2 intervals to predict the test set we receive the following result.

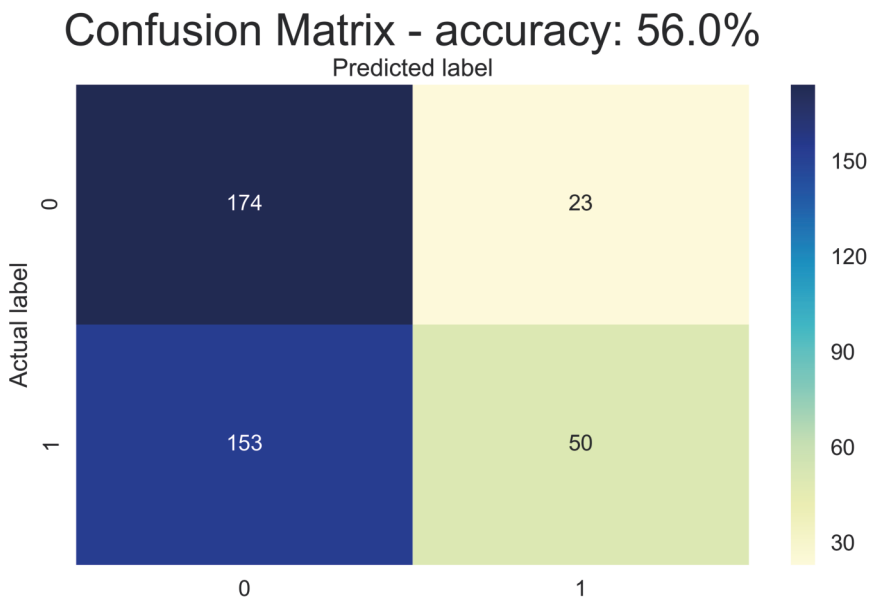


Figure .19:: Predicting the test set with the 2 intervals

D.4: Citric Acid

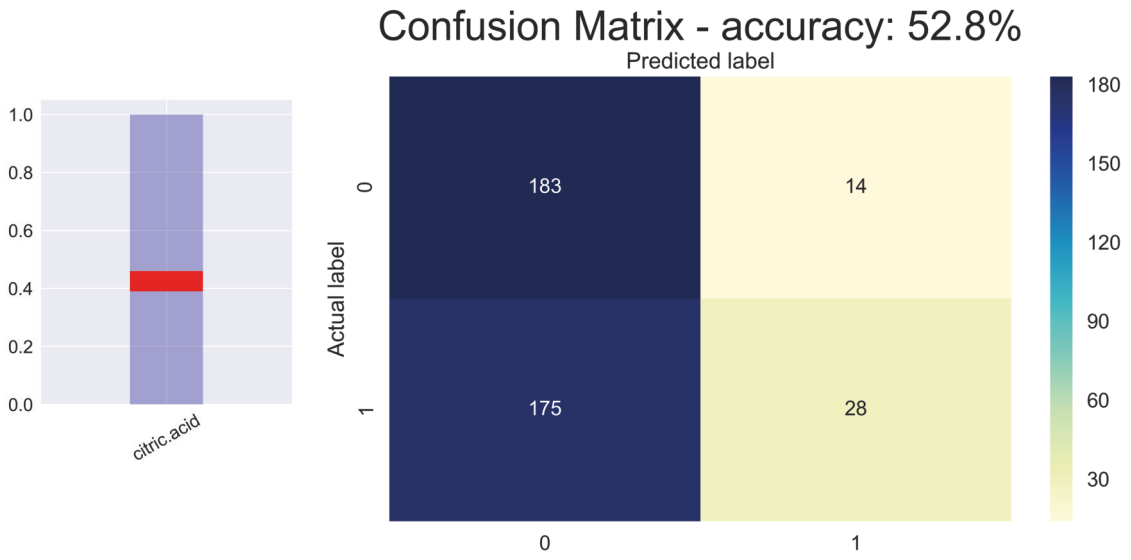


Figure .20:: Interval 2 (decision tree - gini): 139 wines, 103 good and 36 bad

Affirmation

(a) Hereby I affirm that I wrote the present thesis without any inadmissible help by a third party and without using any other means than indicated. Thoughts that were taken directly or indirectly from other sources are indicated as such. This thesis has not been presented to any other examination board in this or a similar form, neither in this nor in any other country.

(b) The present thesis has been produced since March 2015 at the Institut für Algebra, Department of Mathematics, Faculty of Science, TU Dresden under the supervision of Prof. Stefan E. Schmidt.

(c) There have been no prior attempts to obtain a PhD at any university.

(d) I accept the requirements for obtaining a PhD (Promotionsordnung) of the Faculty of Science of the TU Dresden, issued February 23, 2011 with the changes in effect since June 15, 2011 and June 18, 2014 and May 23, 2018.

Versicherung

(a) Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

(b) Die vorliegende Dissertation wurde seit März 2015 am Institut für Algebra, Fachrichtung Mathematik, Fakultät Mathematik und Naturwissenschaften, Technische Universität Dresden unter der Betreuung von Prof. Stefan E. Schmidt angefertigt.

(c) Es wurden zuvor keine Promotionsvorhaben unternommen.

(d) Ich erkenne die Promotionsordnung der Fakultät Mathematik und Naturwissenschaften der TU Dresden vom 23. Februar 2011, in der geänderten Fassung mit Gültigkeit vom 15. Juni 2011, 18. Juni 2014 und vom 23. Mai 2018 an.

Dresden, 09. März 2021